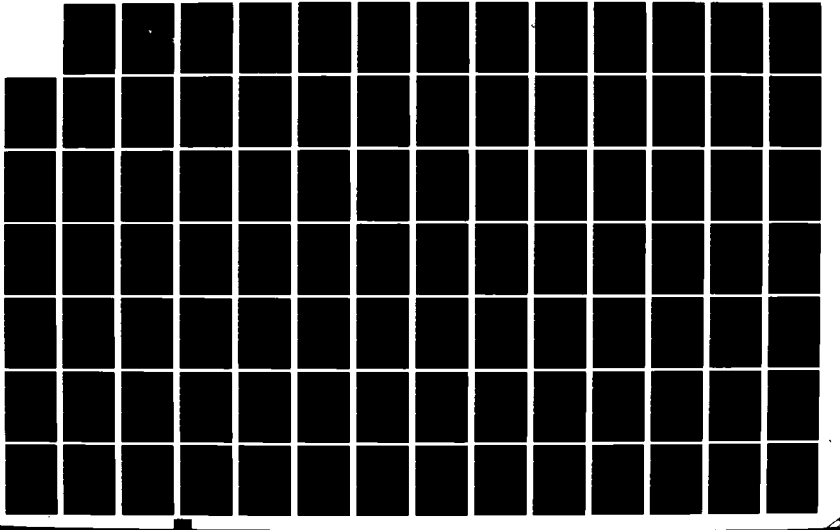
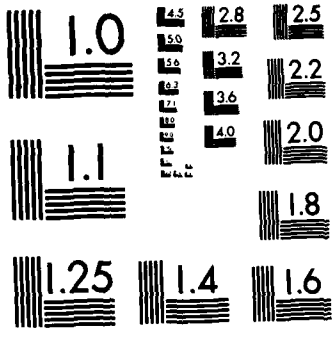
AD-A126 358 SOFTWARE DEVELOPMENT PROJECTS: ESTIMATION OF COST AND EFFORT (A MANAGER'S DIGEST)(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA C J PIERCE ET AL. DEC 82 1/2

UNCLASSIFIED F/G 9/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SOFTWARE DEVELOPMENT PROJECTS:
ESTIMATION OF COST AND EFFORT
(A MANAGER'S DIGEST)

by

Charles James Pierce, Jr.

and

Rebecca Louise Wagner

December 1982

Thesis Advisor: Bradford D. Mercer

Approved for public release; distribution unlimited

DISCLAIMER NOTICE

THIS DOCUMENT IS BEST QUALITY PRACTICABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SOFTWARE DEVELOPMENT PROJECTS: ESTIMATION OF COST AND EFFORT (A MANAGER'S DIGEST)		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis December, 1982 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Charles James Pierce Rebecca Louise Wagner		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE December, 1982 13. NUMBER OF PAGES 103
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Lifecycle
Cost Estimation
Effort Estimation
Software Cost and Effort Estimation Models

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

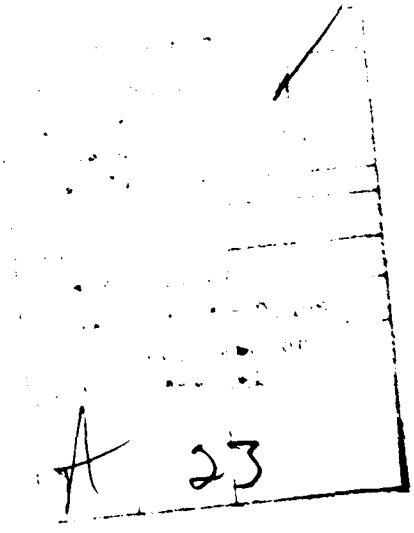
This research focuses on the principles upon which models have been, and may be, constructed for estimating cost and effort in software development projects. A definition of and factors influencing software engineering economics is presented. The major phases and activities of the software lifecycle are described. Effort, time and cost estimation is analyzed. A presentation is then given of some widely used models for estimating cost and effort. Critical factors which must be considered

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (continued)

when constructing a model for estimating cost and effort in software development projects are then presented. We summarize by citing areas that require more attention if cost and effort estimates are to be further improved.



Approved for public release; distribution unlimited.

Software Development Projects:
Estimation of Cost and Effort
(A Manager's Digest)

by

Charles James Pierce, Jr.
Lieutenant, United States Navy
B.A., Queens College of The City University of New York, 1971

and

Rebecca Louise Wagner
Lieutenant, United States Navy
B.A., Bemidji State University, 1977

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN INFORMATION SYSTEMS

from the

NAVAL POSTGRADUATE SCHOOL
December 1982

Authors: Charles James Pierce, Jr.

Rebecca Louise Wagner

Approved by: Barry C. D. Mann
Thesis Advisor

Roger Weissinger-Baylon
Second Reader

Carl R. Jones
Chairman, Department of Administrative Sciences

W. M. Woods
Dean of Information and Policy Sciences

ABSTRACT

This research focuses on the principles upon which models have been, and may be, constructed for estimating cost and effort in software development projects. A definition of and factors influencing software engineering economics is presented. The major phases and activities of the software lifecycle are described. Effort, time and cost estimation is analyzed. A presentation is then given of some widely used models for estimating cost and effort. Critical factors which must be considered when constructing a model for estimating cost and effort in software development projects are then presented. We summarize by citing areas that require more attention if cost and effort estimates are to be further improved.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

A. BACKGROUND

The history of software engineering is replete with tales of projects that have never been completed or have reached completion only after numerous cost overruns and well beyond the originally scheduled operational date. As the problems with software engineering became increasingly apparent, researchers directed their attention to finding ways to more accurately predict the cost, effort and schedule that a software development project would require. Attention has been devoted to determining sound estimates as early as possible in the project. Initially models were developed to provide single estimates in specific environments. Models gradually evolved that could be used at various stages of the lifecycle. Models are now available that can make predictions throughout the lifecycle and can be transported to different environments.

B. PROBLEM

The problem to be addressed in this study is to find those influences that affect estimates of cost and effort in a software development environment. The characteristics that are identified will not necessarily apply to all environments but must be evaluated to determine whether they are contributors to cost, effort and scheduling in a particular situation.

C. GENERAL PROCEDURE

The procedure that has been used was to research literature concerning cost and effort estimations in software development projects. Information was gathered concerning some of the most widely used and successful estimating models. We gathered from this research numerous criteria that must be considered by the estimator before implementing any model that estimates cost and effort in a software development project. We also noted influences on software development projects that have not yet been adequately addressed in cost and effort estimating efforts.

D. ORGANIZATION

Chapter II develops a definition of software engineering economics and presents the major influences on software projects. The software lifecycle is then examined in chapter III in reference to those phases that place the greatest demands on resources. Chapter IV examines the factors important in effort, time and cost estimation. Chapter V presents a number of popular models that have been and currently are in use in estimating cost and effort. Key factors affecting software cost and effort estimation are then presented by the authors of this research paper in hope that these will be addressed in developing a superior cost and effort estimating model.

II. UNDERSTANDING SOFTWARE ENGINEERING ECONOMICS

A. A DEFINITION OF SOFTWARE ENGINEERING ECONOMICS

The term software engineering has been used extensively throughout literature to refer to the various stages of software development and maintenance. Software now commands the major part of any budget for a computer system. In the mid 1950's, 85% of a computer project's budget was devoted to hardware with the remaining 15% given to software. Today, these figures are reversed. [Ref. 1: p. 41] The refinements and advances in hardware combined with the ever decreasing costs of its production have turned focus on software and its ability to exploit the system's innate potential. The financial prominance of software in any computer system demands that whenever we speak of software engineering, we consider the economic impact of our task. Hence, the term software engineering economics will be used in this research paper to refer to the development and maintenance of software.

That we are only now beginning to clearly understand the complexity of the software issue can be seen from the numerous failed attempts to forecast the cost and effort of software development projects. Disastrous software development projects have motivated the development of numerous cost and effort estimating models that have met with varying degrees of success in accurately predicting the course of a software development effort. Successful models have been used as foundations upon which even more accurate models have been developed. The majority of models that are available to estimate cost and effort were developed by private companies to be used in their own working environment.

These models when applied to other environments are unpredictable and therefore of questionable value [Ref. 2: p. 116]. We will examine the most prominent of the numerous cost estimating models and evaluate their characteristics and applicability. We will seek to uncover the remaining problems that currently available cost and effort estimating models inadequately address or completely ignore.

We begin by developing a definition of software engineering economics through reviewing definitions of the term software engineering as offered by a number of prominent individuals in the computer industry. The most comprehensive work on software engineering economics is a recently published text of the same title by Barry Boehm. Boehm defines software engineering as "...the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures, and associated documentation" [Ref. 3: p. 16]. Peters and Tripp at the 3rd International Conference on Software Engineering define software engineering by identifying the concepts and their relationships that surface in a study of software engineering [Ref. 4: p. 63]. Remus of IBM's Santa Teresa Laboratory defines software engineering as "...the science of implementing given functional and performance requirements in a program with optimum quality, at minimum cost, while meeting committed schedules" [Ref. 5: p. 267]. Kerola and Freeman at the 5th International Conference on Software Engineering present software engineering as "...the application of methods, tools and techniques to actions in a reliable and predictable manner or (a) set of stated, technical, economic and social goals for a software artifact" [Ref. 6: p. 91]. We especially note the reference to the social aspects of software engineering. If the human aspects of software engineering are not taken into account as concerning both

the developers and the users, the software product will not realize its full potential. We define the term software engineering economics as the art and science of utilizing analytical techniques, managerial principles and common sense to affectively and efficiently conclude the development and maintenance of software at minimal cost.

B. INFLUENCES ON SOFTWARE ENGINEERING ECONOMICS

1. Size

A number of methods have been used to estimate the size of software development projects. Early estimates of project size are not likely to be very accurate as the exact nature and scope of the project are not conclusively known. Putnam and Fitzsimmons recommend estimating the size of a software development project using the laws of statistics and probability and including the standard deviation for each estimate. Early estimates are based on past experience and the available information the developers have about the project.

As more and more attention is being given to the early determination of the design and specifications of a project, estimators have an increasingly large amount of information to use. The increased effort being given to the front-end development of a project will substantially decrease the final cost and effort expended on a project because of better project preparation. Structural decomposition is used to more clearly understand and closely estimate the size of a project by understanding and estimating the size of each segment of the project. During the development process, iterations of size estimations continue to improve the certainty of the size of the project. Accurately estimating size is the major obstacle in estimating the cost and effort required in software development projects.

The following criteria have been used extensively in estimating the size of a software project: lines of source code and executable instructions. A fairly recent development in complexity estimation developed by Halstead that will be discussed later asserts that size is a function of the vocabulary of a program. The vocabulary of the program is the sum of the operators and operands used. According to the author, lines of code, length (sum of the number of times operators and operands are used) and vocabulary are all valid measures of program size. The problem with Halstead's and other techniques of size measuring is that they are after the fact tools, i.e., the developed software must be available to use them. Although refinements continue to be made in the area of estimating program size, no absolute method has yet been developed that will conclusively estimate size early on.

As the size of a project increases, other factors become more prominent as cost drivers. Complexity, interfaces and the number of people involved become the primary cost drivers. As the size of the project increases, the number of people involved in the project increases and significant new problems are created. Brooks learned from his experience with the IBM OS/360 project that men and months are not interchangeable. Using man-months to measure the size of a project is dangerous since men and months are only interchangeable in an environment where a job can be perfectly partitioned among workers and workers separated from each other to preclude communication. In reality, training and communication take up a significant amount of time in increasingly large projects. [Ref. 7: pp. 13-26] And although time consuming, communication is essential for a successful project. Esterling's research also showed that project completion time can be improved upon only up to a certain point by adding personnel. Added personnel eventually serve only to delay the project. [Ref. 8: p. 168]

2. Complexity

Software engineers use complexity to denote treatability, maintainability, readability and/or comprehensibility of a program [Ref. 9: p. 317]. Complexity plays an important part in two phases of a software lifecycle: development and maintenance. The complexity of a program will directly influence the cost and effort in testing and debugging and in correcting bugs that subsequently surface from use. The difficulty of modifying a program due to changing requirements will also be directly related to the complexity of a program. Complexity measures have proven difficult to objectify in project evaluations. The main problem with both size and complexity measures is that they are done after the fact, i.e., after the code has been written. A complexity measure will be judged on its ability to predict programmer performance. Much research in the complexity area implies that programmer performance can be predicted from the source code of a program.

The question being asked today is which factors of the many researched in programs best capture program complexity. Two other factors have shown to influence program complexity: the programmer and the programming task. Significant individual differences have been found in programmer performance. "The important point here is not that individual differences among programmers exist, but that the variability is so large that experimental results may depend more on individual differences than on experimentally induced differences" [Ref. 9: p. 317]. What might be very difficult for one programmer may be easy for another thus nullifying the value of that predictor. Programmer performance must be based on a combination of program related complexity measures, programmer traits and programmer tasks.

One of the newest approaches to measuring complexity has been presented by Halstead in which lines of code are broken down into operators and operands. Three advantages of this approach are:

1. An explainable methodology for calibrating a measurement instrument.
2. A more nearly universal measure, since the approach is consistent across the boundaries of programming languages.
3. The ability to relate some of the effects of programming style to measured quantities. [Ref. 10: p. 373]

The rules for this method seem to combine lines of code, decision nodes and operation codes, variables and punctuation. The emphasis given to each area is questionable but at least they are all included. [Ref. 10: p. 374]

Halstead defines length as a function of sum of operator usage and operand usage. Length can be estimated from vocabulary with reasonable certainty according to Halstead. Volume is a function of vocabulary and length. Lines of code, length and volume are equally valid as relative measures of program size. Program size measured in lines of code, length or volume is a function of vocabulary.

Halstead also presents an equation for measuring difficulty. Difficulty is defined as the measure of ease of reading and ease of writing a program. Difficulty affects the effort needed to code an algorithm, to inspect and review it and to evaluate it later when changes need to be made to it. Various levels of difficulty are experienced due to the skill level of the programmer, poor program structure or the lack of experience with a language and possibly the complexity of the algorithm. [Ref. 10: p. 381] Halstead identifies six code impurities that if eliminated reduce the level of complexity of the program. They are as follows:

1. Complementary Operations: unreduced expressions
2. Ambiguous Operands: the same variable means different things
3. Synonymous Operands: giving the same value to more than one variable name
4. Common Subexpressions: subexpressions used more than once in a program. The subexpression should be given a unique variable name
5. Unwarranted Assignments: assignment of a variable to a subexpression even though the variable is used only once in the program
6. Unfactored Expressions: easy to understand but at times hard to follow in coding. [Ref. 10: pp. 382-383]

Halstead's measures are attractive in that they are easy to automate.

Another measure of complexity that has achieved some measure of universal acceptance is that presented by T. McCabe. In McCabe's cyclomatic complexity measure, all the decision points in the Procedure Division of a program are counted, those for each paragraph and section are summed, and those for the entire program are summed. A paragraph is assumed to be the size of a module and assigned a complexity value of one to start. When a complex conditional statement is encountered, each simple conditional expression is assigned a value of one. Research to correlate Halstead's and McCabe's measures with programming effort have shown the following (especially respecting Halstead's work):

1. Reasonable correlation exists between the measures and programming effort.
2. A comparable correlation exists between the measures and the number of instructions in the programs.
3. Number of instructions seem to be as good an indicator of software development effort required for

large programs (over 1000 DSI (delivered source instructions)) as the Halstead and McCabe measures. The measures, however, correlate better than DSI with th amount of terminal time required to program small programs. [Ref. 3: p. 481]

The weaknesses of the measures lie in their not accounting for such factors as personnel experience, hardware constraints, managerial factors and the use of tools and modern programming practices. The user must also become accustomed to using the measures and, as already stated, the measures involve a knowledge of program characteristics that are not learned until the program is written.

3. Interference

Interference factors include the total of all disturbances that affect programmers' productivity. Administrative or non-direct work includes such activities as budget preparation, union meetings, and status report submissions. Social interactions are a second source of time loss. Thirdly, interference includes the time consumed in regaining a creative thought pattern after interruption. Creative people are subject to environmental influences on their ability to evolve a new program. A fourth source of interference is the time spent coordinating with other programmers while developing a program. A fifth source of interference is the number of miscellaneous interruptions that result from passing social interactions, trips to the head, etc. [Ref. 8: pp. 164-166]

Intercommunication is essential to any project. To minimize intercommunication, as few people as possible should be involved in a large project if completion time is important (as inevitably it is). Brooks suggests the use of programmer teams to improve upon the completion time of a project. The task is divided up into a number of segments

and each team operates on its own as far as possible to complete a segment of the project. Esterling's research showed that programmer productivity can be increased in an interrupt free environment. Interference factors command a large portion of the programmer's time and must be addressed in any estimation of cost and effort.

4. Cost

Cost has played a major role in developing software engineering economics due to the many cost overruns on software engineering projects. Cost overruns have become the driving factor in efforts to develop software cost and effort estimating techniques. Escalating personnel costs have driven companies to new awareness of software development projects. A severe shortage of software engineers presently exists along with greater shortages in the number of senior software engineers whose competence and expertise in guiding a project can often result in an outstanding product as opposed to a mediocre product. The job market for software engineers is good and the cost of hiring programmers and analysts continues to grow in dominance in the overall cost picture. Estimates indicate that the cost per man-year of a software engineer will be 100,000 dollars by the mid 1980's (this includes salary, fringe benefits and support costs).

Software projects will usually take at least two to three years to complete. One programmer will usually not suffice to complete a project so a number of salaried software engineers must be anticipated. But as already discussed, adding programmers to accelerate a software development project will only be beneficial up to a certain point beyond which diminishing returns will be realized.

Initial development cost may be expensive for a project but experience indicates that for every five dollars spent on initial development, between seven and twenty dollars will be spent on maintenance. With this skyrocketing picture of costs throughout the lifecycle of a project, estimates for a software development project and the subsequent plans for and implementation of a software development project must be carefully managed. Since so much of costs will involve personnel, software development environments will be increasingly looked to for the best ways to exploit the potential of software engineers. [Ref. 11: p. 227]

Recent findings indicate that contrary to intuitive feelings about the matter, the total cost of a project will decrease along with development time when overtime is paid to workers. If time and a half is paid, the overall cost decreases; if double time is paid, the overall cost remains constant. Indirect costs will have a separate impact on overtime work since they do not vary over time. If the indirect costs are high, savings can be realized by hiring consultants and by-the-hour people. [Ref. 8: p. 170]

Thus we see that the primary driver of the cost of a software development project is the personnel involved. Personnel must be carefully selected for a particular software development project. As will be discussed later, past experience of the programmer is of considerable importance. After personnel are selected for a development project, the management process implemented will determine how fully their collective potential is exploited.

5. Quality

The quality desired in a given software product will directly influence the cost and effort devoted to the project. Quality will generally vary according to the

nature of the project. Software developed for a manned lunar flight will of necessity be of far greater quality than that to support standard business applications.

Remus defines quality as "...the number of program defects normalized by size over time" [Ref. 5: p. 268]. We find this to be a useful, working definition of quality. Quality of a software product can be improved by increased attention given to the front-end design process with emphasis on modularization. Modularization or dividing the project into small segments that are more intelligible enables the programmer to more easily understand the objective of a task assigned. A better understood assignment will lead to a better product.

Programming environment has a significant impact on quality. The ability of the programmers to work in an environment conducive to and supportive of creative thought will foster a superior software product.

The cost of quality software will not go down as dramatically as the cost of hardware [Ref. 11: p. 226]. Very cheap, unwarranted, unsupported software will appear on the market and be available to the consumer. Inexpensive, mass marketed, supported software is not a practical possibility for the future. Four types of software products will be available in the future:

1. Quality products requiring no support and known to be correct and to function predictably and reliably
2. Quality products that are sold to customers willing to pay the support costs
3. Custom-made products, developed for a specific user's needs
4. The others. [Ref. 11: p. 227]

Prices for type 1 products will be high and vary according to market demand. Type 2 products will be priced considerably higher than type 1 products. Type 3 products will be the highest priced of all software. Type 4 products will be moderately priced for mass consumption. Especially sophisticated software will be sold along with associated hardware in what will be a turnkey system.

6. Scheduling

Scheduling is important in software development projects so as to avoid slow down in a program due to the lack of coordination among interdependent segments of the project. Scheduling shows where in time all important project events take place. The schedule should include milestones, reviews, key meetings, audits, documentation releases and product delivery dates.

Scheduling is also important for marketing and sales purposes. A product must be available at the time when the marketing personnel have promised it. The bottom line for any organization is customer satisfaction and hence profit.

Project management differs from production management in the nature of the task. Production management involves the performance of a repetitive job. Project management is much more difficult in that the job to be performed and the results of the effort are not clearly understood at the outset and are unique for the most part. The following characteristics apply to all projects in varying degrees:

1. The project itself will last for weeks, months or even years. During this time, many changes may occur in the project which may affect cost, technology and resources.
2. The project is usually complex involving many interrelated activities that must be monitored.

3. Projects are expected to be completed on time with any delays costing the developers into thousands of dollars per day of delay. Not only is money lost but also much ill will may be created from overdue projects.
4. Projects often are sequential in nature with the start of one project dependent on the completion of another. [Ref. 12: p. 273]

As a result of the nature of projects, planning, control and coordination of projects is a complicated task that requires close attention. Until recently, no formal, generally applicable method was available to manage the progress of projects. Two methods are now available that have proven to be very useful in project management: PERT (Program Evaluation and Review Technique) and CPM (Critical Path Method).

Two differences exist between PERT and CPM. The first involves estimating activity durations. An activity is an effort that consumes resources and a certain amount of time. PERT uses the weighted average of three estimates in order to arrive at an expected completion time based on a probability distribution of completion times. Because of this, PERT is looked upon as a probabilistic tool. CPM is a deterministic tool, i.e., only one estimate is made for duration of an activity. The second difference between the two methods is that CPM can give an estimate of costs as well as completion time for a project. PERT is fundamentally a tool to plan and control time; CPM is a tool that can be used to plan and control both time and cost of a project.

PERT and CPM attempt to answer the following questions:

1. Which activities are critical? That is, which must be completed on time to keep the project on schedule?
2. Which activities are noncritical?
3. How much flexibility does management have in executing the noncritical activities?
4. What is the earliest expected completion date for the project?
5. What is the best way to handle delays that are detected during execution of the project? [Ref. 12: pp. 274-275]

In addition, PERT answers the following questions:

1. What is the chance of completing a project by a desired date?
2. For how long should a project be planned so that a given probability of completion is attained? [Ref. 12: pp. 274-275]

CPM answers the following additional questions:

1. What is the least-cost way to expedite the completion of a project?
2. What is the shortest possible time for a project to be completed? [Ref. 12: pp. 274-275]

PERT and CPM provide numerous advantages for the project manager. The requirements of the methods force managers to plan ahead in detail to determine what has to be done to meet project objectives on schedule. Definite decisions must be made regarding execution times and completion times for activities in the project. The tools of CPM and PERT provide for improved communication among departments in the organization and between the developer and clients. The devices allow for identifying critical activities in the project and thus close attention can be given to these phases. Since critical activities are most likely to be

potential problem areas, these difficulties can be spotted early and adequately planned for.

Resources are more easily managed using PERT and CPM. Once bottlenecks and problems are identified in the project, resources can be more easily moved around to correct difficulties. Deviations from schedules are more easily identified and accommodated. Since PERT and CPM provide an overall picture of the project, the tools can be used easily to present the project to lower levels of management. PERT and CPM are easily adapted to computers. Alternate ways of executing projects can be evaluated using PERT and CPM. PERT provides the probability of completing a project on schedule while CPM allows management to evaluate the costs of rushing activities. Many scheduling problems can be avoided through close adherence to management tools like PERT and CPM.

Again we observe that attention to the front-end development of a project will add immeasurably to its smooth accomplishment. The ability to adhere to a schedule will additionally contribute to a project's success as the employees will realize personal gratification as milestones are met. Improved motivation will mean an improved product.

7. Past Experience

Past experience plays a significant role in software development projects. Companies that have past experience in large jobs will tend to overestimate a job and manage the job as a large job. Companies with experience in small jobs will tend to underestimate a job and manage it as a small job. This entire concept has been neglected in each cost and effort estimating model reviewed by these researchers. **[Ref. 13: p. 43]**

Research has found that experience is important if the experience that a programmer has is related to the current project. Merely programming for a number of years will not mean that someone is a good programmer, only that he has been programming for a number of years. He may have been making the same mistakes and using the same procedures during those years. So the developmental pattern of the individual programmer and analyst must be examined in order to ascertain the maturity of the individual. Programmer productivity varies greatly on the same task, some research reporting variation of 5:1 while other research has found variation up to 20:1. Literature on programmers' experience will be addressed again in another segment of this paper.

3. Tools

Software tools have become increasingly a topic of research in this decade as software has become so dominant a factor in the development of computer systems. The ergonomics of software engineering has been described as "...the discipline of analyzing and understanding the requirements for quality software engineering tools, and of translating this understanding into innovative tool design" [Ref. 11: p. 223].

Ergonomics deals with the mutual adjustment of man and machine. Man has done most of the adjusting as of this time and machines now must adjust to human needs. This evolution has come about due to the increased costs of hiring and supporting programmers. Man initially exerted all efforts to explcit computer capabilities; now, computers must evolve to exploit human potential. The easier software development tools are to use and the more affective they are in assisting the programmer to produce his product the more efficient will be the entire development program.

The tools used during the production process can be divided into a number of groups.

1. The design language should be general enough to permit a description in general terms and specific enough to be unambiguous. Analyzers assist in finding obvious problems and automate some interconnection cross references. Tools such as the Problem Statement Language/Problem Statement Analyzer are computer-aided structured documentation and analysis techniques that aid in developing the requirements and specifications for a program and in the formulation of documentation as the project proceeds.
2. Editors and on-line document handling facilities allow machine use for writing, producing and maintaining specifications and user publications.
3. Code library facilities improve testing and integration of fixes for code errors.
4. A data dictionary system, a software system used to record, store and process information about all of a firm's significant data entities and related data processing functions, provides the following benefits:
   a) Security and access control for data base environments
   b) Standardization of data elements
   c) Identifies redundancies in the data base
   d) Automatic documentation with current information
   e) Improved transportability between computing environments
   f) Assists auditing [Ref. 14]
   g) Interactive code facilitates program development allowing each programmer to use a terminal in his work

h) Test simulators allow simulation of complicated hardware configurations

i) Test control and test case libraries facilitate testing procedures

j) Service data bases provide solutions to errors found that are not yet corrected for public usage. [Ref. 5: pp. 273-274]

Software Development Environment (SDE) is the name now used to describe the tools available to programmers to develop a software product and to maintain it. SDE's can be as simple as a mixture of assorted tools with little direct relation to one another, or as sophisicated as a particular development methodology using tools or software utilities that are highly integrated and non-repetitious. [Ref. 15: p. 20] SDE is a recently developed concept.

> It appears the software development environment should be adaptable, user-centered, suggestive, helpful and supportive, not imposing. The tools of the environment should be portable, methodology independent, catalogued with respect to assumed user sophistication and they should have a specific purpose. Finally, the environment should support large-scale software production and provide a consistent interface through the entire software life cycle. [Ref. 15: p. 21]

SDE should provide tools that are integrated and user friendly. User friendly characteristics should include such things as human interfaces other than text, such as menu selection capability, graphics and possibly voice recognition. Not much concern has been shown up to now as to the cost of implementing such environments or the cost of sustaining such environments [Ref. 15: p. 21].

Common potential benefits to be derived from the use of SDE include improved software quality, reduced cost of software, improved programmer productivity, and more management visibility. The prevalent feeling is that the use of software tools and the SDE is good but as of now no experimental data exists to corroborate these feelings.

The cost of SDE has not been closely studied as the environments have been developed to support large systems and these systems are usually used by large organizations that have substantial resources. Most of the effort is directed toward supporting the development phase and not the maintenance phase. Companies feel that the development cost will be shortened and therefore support the SDE. Not much attention is paid to the maintenance phase as maintenance is considered a source of income for the companies. [Ref. 15: p. 24]

We believe that little attention has been given to estimating maintenance costs for the same reason: maintenance is seen as a source of revenue. The SDE is made up of a number of components. The software development tools and in some cases an implicit set of operating procedures are generally understood to be part of the SDE. The SDE also includes the organization that is supporting the environment and the integration of the SDE with the corporation as a whole. An SDE integrated with the corporation as a whole is important for the proper functioning and utilization of the environment.

An automated software development environment requires sophisicated software support for complex directories of files, a sophisicated database management system and a standard interactive capability. These capabilities require considerable hardware support.

SDE has had a stated goal of reducing the time to develop software. Studies done by Boehm indicate that the development time is not reduced but that the time spent in development is shifted from writing source code and debugging to developing the requirements and specifications. [Ref. 16] The major problem with the concept of a software development environment is getting companies to allocate necessary funds to its development and support. Hardware,

personnel and training must be provided to implement a software development environment and to maintain its smooth operation in the company.

9. Management Policies

Management by Objectives (MBO) is quite compatible with using PERT and CPM and scheduling methods. "MBO refers to a formal, or moderately formal, set of procedures that begins with goal setting and continues through performance review" [Ref. 17: p. 144]. MBO is a participative process that involves communication among managers and staff members at all levels. Established links of communication facilitate the planning and control of a project. MBO assumes that workers are motivated to perform their jobs and want to do as good a job as possible. This view of human behavior, called Theory Y, is opposed to Theory X, a view that holds workers to be not very reliable and only interested in work as a means of survival. People will avoid work whenever possible according to Theory X.

Programmers are known to be highly motivated individuals who want to create as good a product as possible. They generally are not too interested in other non-scientific people and are mostly concerned about exploiting the fullest potential of the computer. A sharp program manager will recognize the needs of his programmers, meet those needs to allow the programmers to produce their best product, and insure a cooperative climate exists among programmers and programming teams and groups. The critical role of a program manager will be more closely addressed later in the research.

MBO involves primarily the establishment of goals through a joint effort of management and subordinates. Objective measures of performance are arrived at, i.e., lines of source code generated. Performance reviews and

regular periodic reviews are made. A primary purpose of MBO is to achieve efficient operation of an organization through the efficient operation and coordination of its parts. It has great value in performance planning and appraisal. Managers in the organization are encouraged to work with personnel above and below them in an effort to achieve the best product possible. When problems arise, the team works together to solve them rather than to seek someone to hang. Since programmers are creative people, progressive management policies like MBO emphasizing the goals of self-actualization are encouraged.

10. The Project Manager

Software development projects are often large scale projects requiring the highest coordination. The qualities that the Federal Government seeks in its program managers are herein presented for their overall application to any large scale, software development project. Oftentimes, government acquisition is the driver behind a software development project. The characteristics of the project manager who guides a software development project to its completion will be critical for the success of the project. Managing an acquisition program for a large scale, government purchase is a demanding task and requires an individual of unique skills and personal character traits. "The accomplishment of this objective requires the successful integration of people, financial and material resources...in one word--Management" [Ref. 18: p. 8]. "A program manager is expected to have an in-depth technical understanding of many areas, to plan, organize, and control with the precision of a military campaigner, to integrate ideas and write 'Like a journalist,' and to build and motivate a team of managers he may have never met before or work with again" [Ref. 19: p. 6]. The responsibility for the success or

failure of the acquisition program lies in the hands of the program manager. The job must be done efficiently, within the budget and on time. The success of the program will be a direct reflection on how well the team has been motivated to achieve its goal.

Even if we know the proper way to build and motivate a project team, more importantly we must find a program manager who can successfully implement this knowledge. Most importantly, a program manager should be an individual with a positive attitude and keen insight into human nature. Successful projects emerge from people who believe that the job can be done regardless of the obstacles. If the program manager is a positive thinker, he will foster this attitude on his team.

An achieving program manager will demand outstanding results. Outstanding effort is admirable but if the product is not delivered as advertised, the effort is empty. If production has been taking an inordinate amount of time on the part of certain individuals, personnel reassignments should be considered. A program manager should be one who remains above interteam squabbles and criticism and be the individual who puts such destructive forces to rest. He should be an individual who is bound by his work, keeps his promises and thereby generates a feeling of confidence and certainty within team members. [Ref. 20]

An effective manager "...must have skill in communications, which spans such areas as the ability to express ideas clearly, the ability to lead discussions and arbitrate differences, the ability to ask the kind of questions that stimulate and encourage creative thinking and problem solving. He must also master the skill of listening---so that he understands what is said and what lies behind the words" [Ref. 21: p. 15].

> A recent study indicated that employees view communications with supervisors as the most satisfying and important relationships in the working environment, but least able to establish. In another study conducted at Loycla University, essential attributes of a good manager were compiled. It was found most important that managers listen well. Since attentive listening is the best way to stay in touch with everything that is happening, such managers are well informed. Good listening, in addition to keeping managers well informed, promotes good human relations. [Ref. 22: pp. 4-6]

A program manager must feel secure within himself. He must be able to function with the knowledge that he will be held personally accountable for the success or failure of the acquisition program and will be dealt with accordingly. Above all else, we feel that a program manager must have a talent for human understanding. He must have insight into behavioral patterns that indicate personal or professional trouble within the staff member. Through personal attention to the needs of the individual, he will generate a loyalty that will motivate the best actions from the indivivual thus improving the person for future achievements and thrusting the current project to a successful completion.

Above all else, the program manager is the key to a project's success. Sound estimates of cost and effort will be for naught if a competent program manager is not at the helm.

III. SOFTWARE LIFECYCLE: MAJOR PHASES AND ACTIVITIES

This chapter describes the major phases and activities of a software development project. With any type of project, whether it be developing a software system or building a little red wagon, a person needs to know exactly what it is he is setting out to do before he can even begin to estimate what he needs in terms of time, money, and effort to complete the project. Throughout the literature on software engineering economics, reference is made to the lifecycle phases of software development projects. Essentially, a project is broken down into parts so that what may at first appear to be an insurmountable task may be viewed as a composite of less complex components. An understanding of the phases and activities involved in the production of software is the first step toward answering the question "Where does the money go?".

A. MAJOR PHASES

1. System Requirements/Feasibility

We will devote considerable attention to this phase of the software lifecycle. Too often we charge off to battle when no war exists. The corporate manager must first determine that a real need exists in his company and that the need can best be satisfied with improved software or initially computerizing an area of his operations. The perceived problems, however, may be found to be solvable within his existing framework.

During the system requirements/feasibility phase, software concepts must be delineated and evaluated and a preferred alternative chosen by management.

> Once the need for a new information system is perceived, a feasibility study determines whether or not desired objectives of a proposed information system can be achieved within existing constraints. The study identifies the cost of proposed changes (monetary and organizational) and estimates the benefits of the new system. On this information, the manager decides whether to implement the new system or discontinue the study. [Ref. 23: p. 233]

A feasibility study is undertaken when the need for a new or better information system is perceived by an organization. A feasibility study is a costly undertaking and before beginning the company should evaluate whether existing solutions to similar or identical problems exist and whether they can be satisfactorily adapted to their own company.

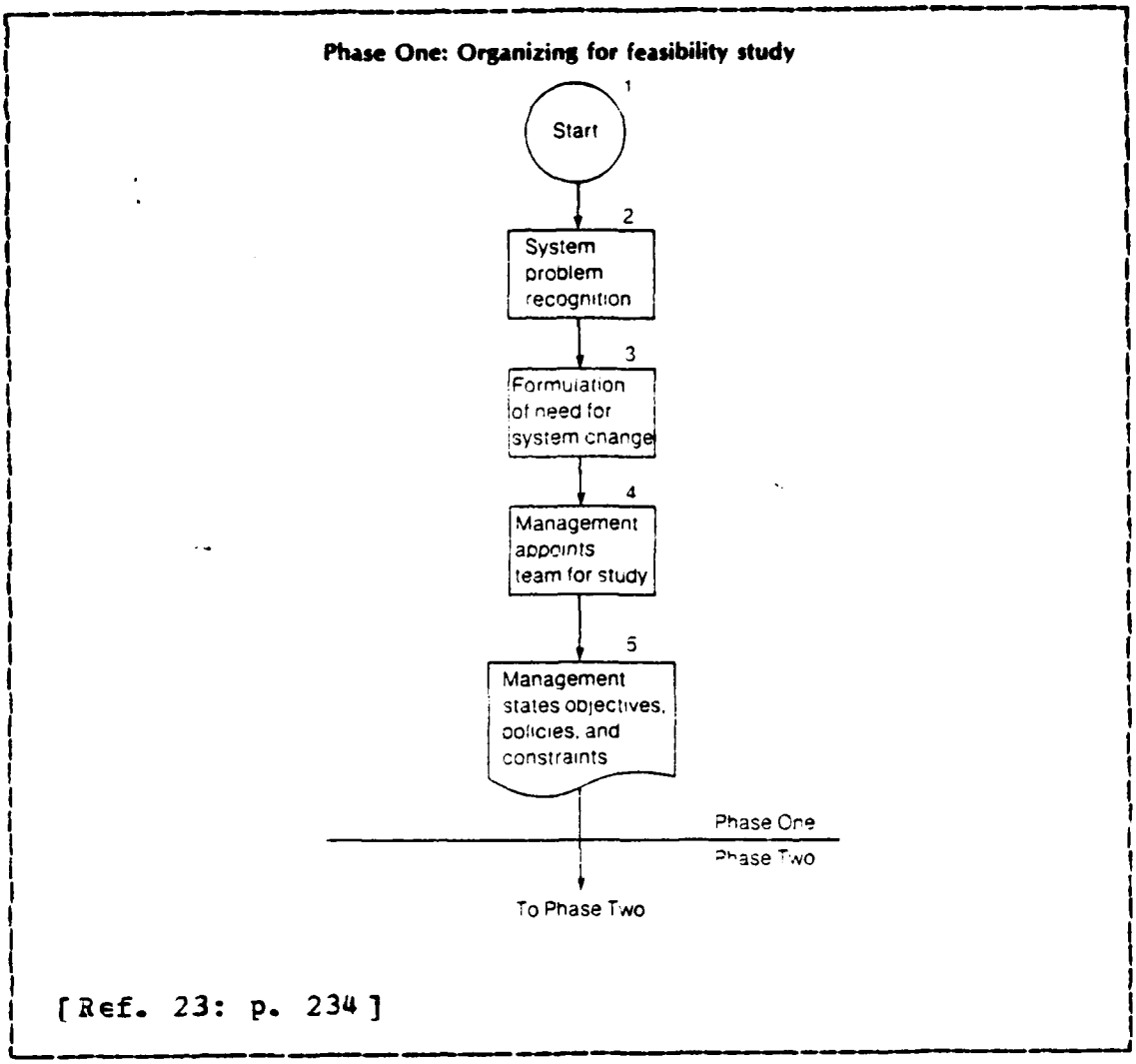
When a software development project is contemplated, the market's existing software should be examined to determine whether the needed wheel has already been invented. In assessing the requirements of a particular software development project, the existing hardware must be reviewed as to whether it can perform up to the expectations and demands of the contemplated system. If the hardware is nonexistent or outdated, the feasibility study must incorporate the areas of hardware and software.

The four phases of a feasibility study are:

1. Organizing for the feasibility study.
2. Search for a solution.
3. Feasibility analysis.
4. Choice of a solution. [Ref. 23: p. 233]

Phase one, organizing for a feasibility study, is undertaken when one or all of the following become apparent:

1. Changes in organizational goals, plans and information requirements.



[Ref. 23: p. 234]

Figure 3.1 Phase One: Organizing for feasibility study.

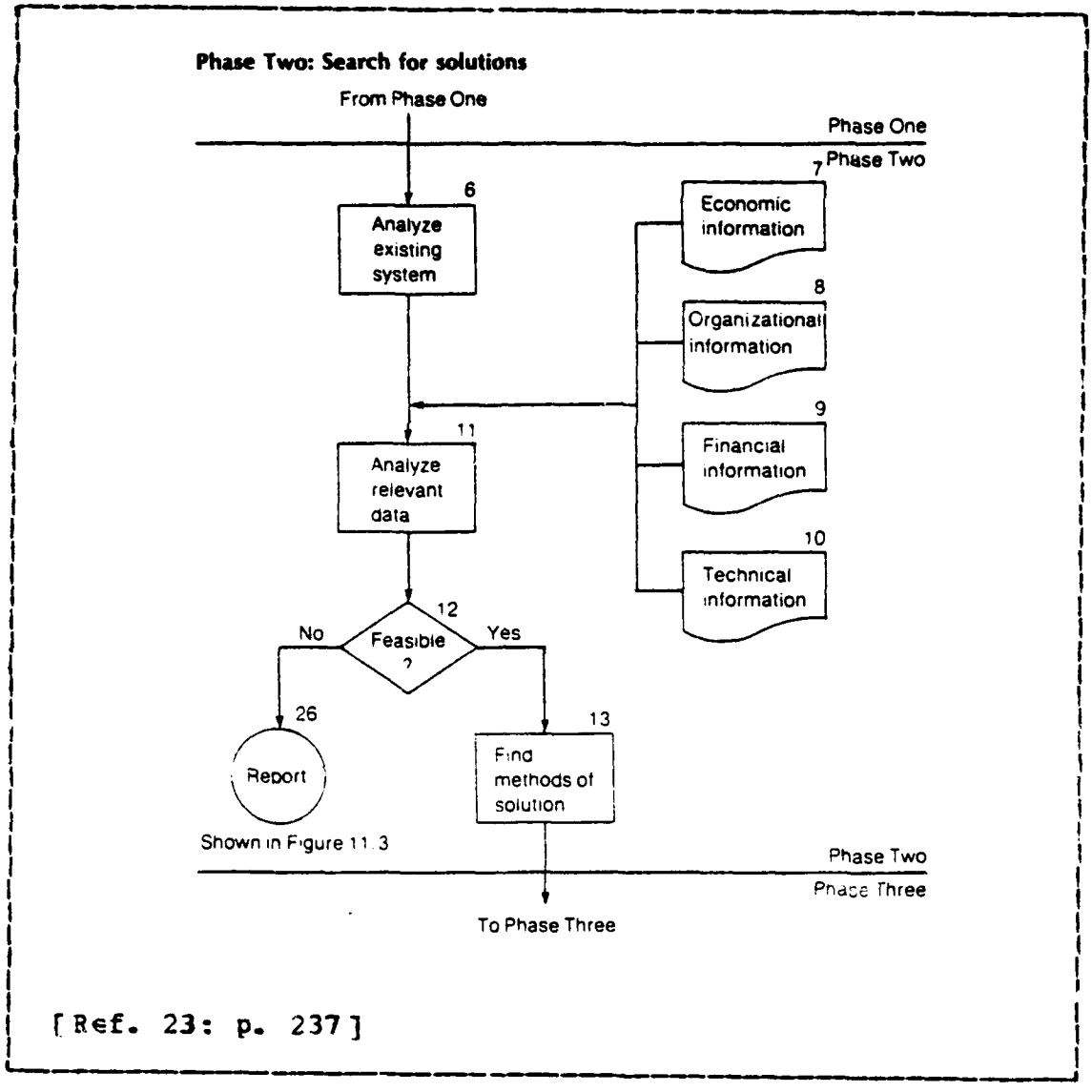
2. Changes in organizational structure (e.g., appointment of new top management).
3. Changes in the environment (e.g., legislation requiring the company to supply new data to government agencies).
4. Changes in technology that may make new systems feasible. [Ref. 23: p. 234]

If the need for change has been clearly identified, then management must undertake to clearly define the problems and search out possible solutions. A feasibility study team is recommended for this task. The team usually consists of two to eight members with the following qualifications:

1. Members should reflect a knowledge of the system techniques. The nature of the problem will determine whether this knowledge be in the area of operations research, statistics, computer science, information science or business functions.
2. Members should have the ability to relate to people since their work will lead them to exchanges with many individuals in the company. Change and possible loss of jobs always concern employees and these fears should be alleviated by the group members.
3. Members should have a thorough understanding of the organization.
4. Members should be able to digest details and relate them to the overall picture of the organization.
5. Members should have a position in management for clout.
6. Members should have experience in the project under consideration. [Ref. 23: p. 235]

Personnel may have to be hired to meet some needed qualifications.

After the team has been identified, management will state the objectives of the study and the related policies and constraints. The team will need to know such things as permissible error rate, how many decimal points answers should be carried to, response time requirements, the number of users anticipated on the system, location of the users, etc. Goals are set by management and the feasibility study is to determine whether the goals can be met within

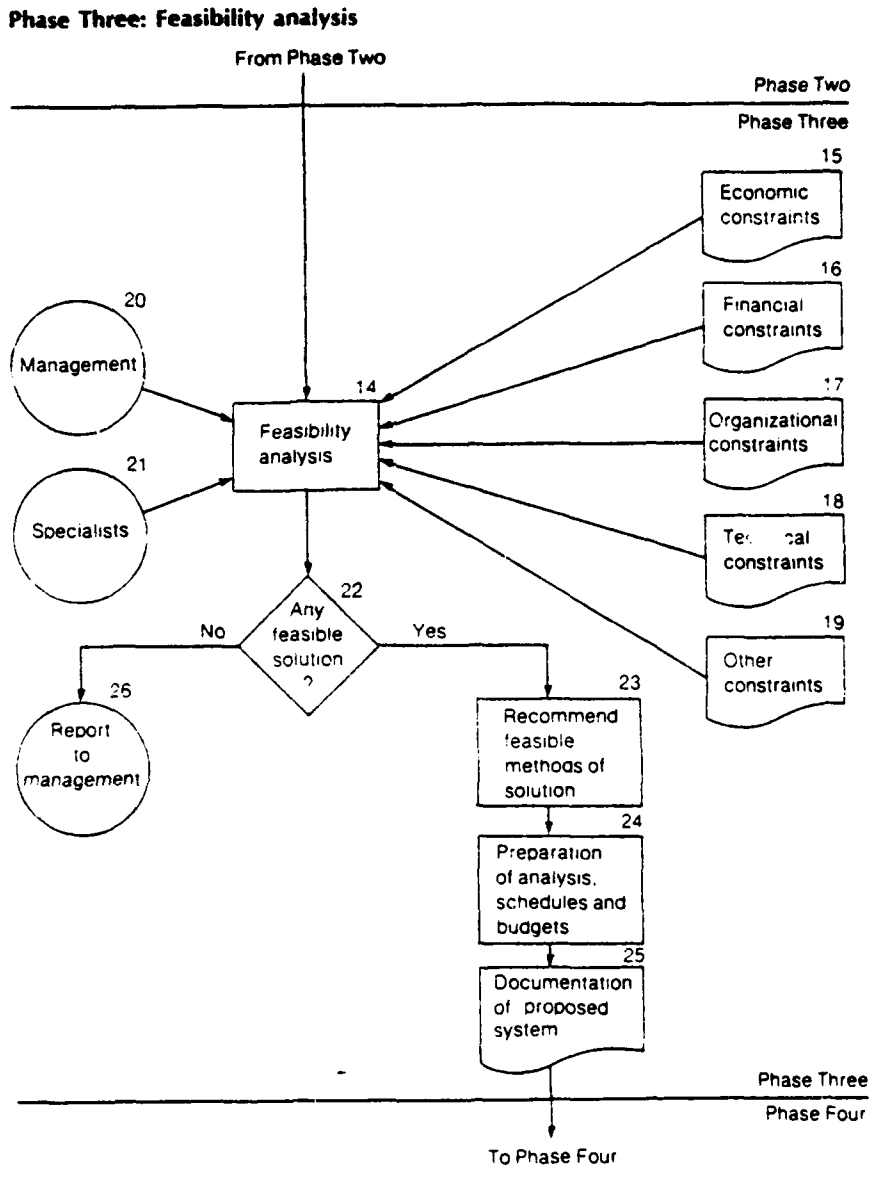


[Ref. 23: p. 237]

Figure 3.2 Phase Two: Search for solutions.

technological constraints and resource constraints of the company. If goals cannot be met as originally defined, either the goals are redefined or the project is scrubbed.

Phase two, the search for solutions, may take two forms. For a situation where major overhuals are to be done on a system, a fresh approach to the problem disregarding



[Ref. 23: p. 239]

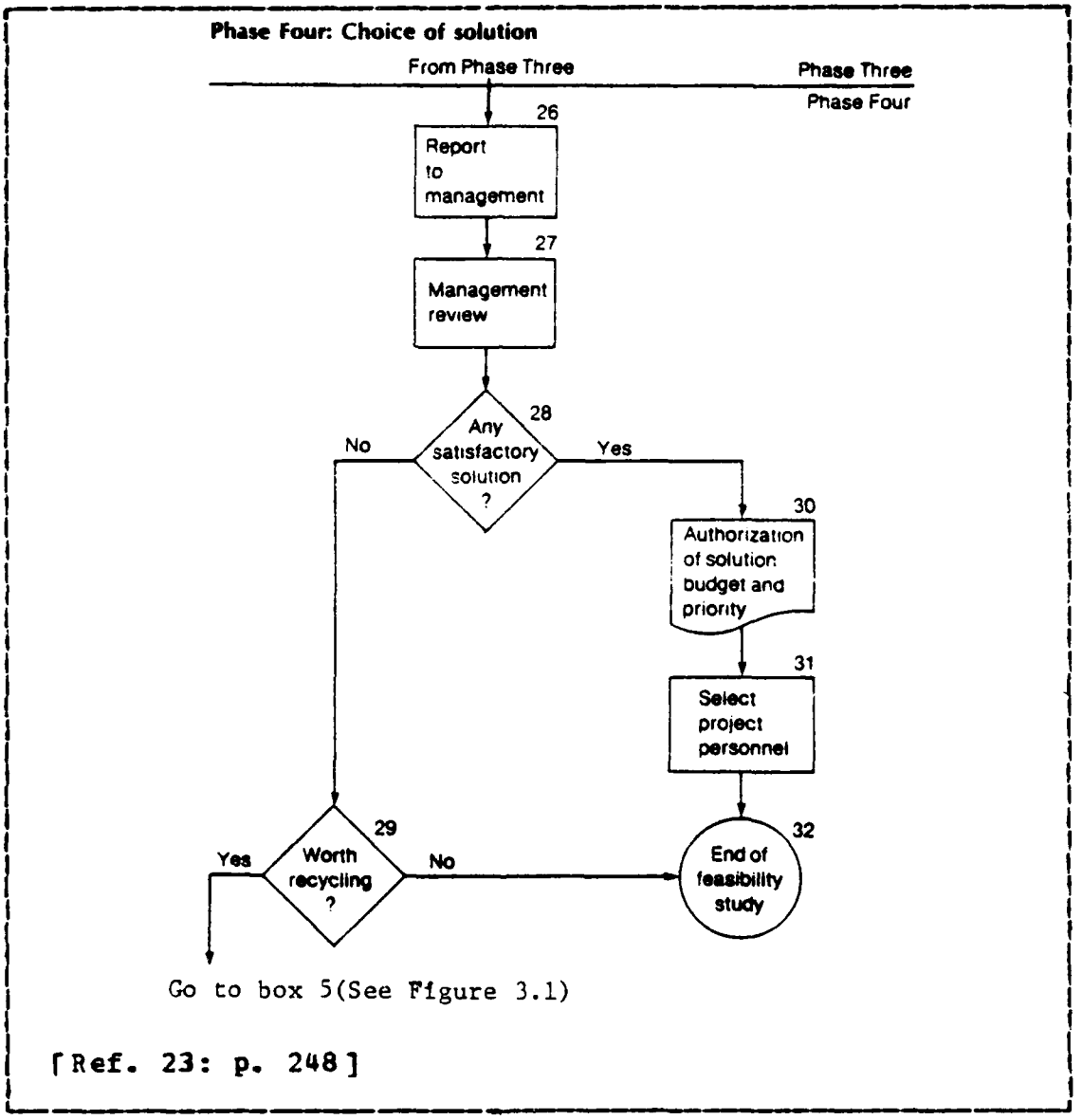
Figure 3.3 Phase Three: Feasibility analysis.

the existing system is recommended. When changes to the subsystems within the existing structures are to be undertaken, then a thorough evaluation of all the information on the environment is recommended so that current performance of the system can be evaluated and changes recommended. [Ref. 23: pp. 237-238]

The solutions uncovered in phase two are tested in phase three, feasibility analysis, regarding their economic, financial, organizational and technical viability considering imposed constraints. The economic feasibility of implementing a new system is usually accomplished by performing a cost-benefit analysis of the proposed undertaking. The cost-benefit analysis will determine whether the benefits of the new system will be greater than the costs required to implement the new system. What must be taken into account are the costs encompassing the software and hardware as required.

Increased attention is being given to organizational adjustments that must be made when a new information system or a revised information system is contemplated. "The major reason Management Information Systems (MIS) have had so many failures and problems is the way systems designers view organizations, their members, and the function of an MIS within them" [Ref. 24: p. 17]. Although management information systems are cited, the authors include any computer based information systems effort. Faulty views of the organization result in a faulty design of the information system and hence a less than optimal operating system. The Socio-Technical System (STS) design approach offers excellent advice on implementing an information system by taking a realistic view of the organization. The feasibility study group would do well to recommend or incorporate ideas from this approach. Both the technical and social aspects of a new system must be considered in the design of the system.

STS is a fairly recent development in the quest for organizational systems which are both more satisfying to their members and more effective in meeting task requirements. This approach is used for redesigning existing work systems as well as for new site designs. [Ref. 24: p. 17]



[Ref. 23: p. 248]

Figure 3.4 Phase Four: Choice of solution.

In phase four, choice of a solution, the feasibility team recommends various alternatives to management with a ranking of their desirability. If no desirable solutions exist, management may want to change their constraints in order to find a feasible solution. Although management will have been involved in the feasibility study as it progresses, it must now make a final review of the alternatives and settle on a choice.

2. Software Requirements

Defining software requirements means defining the aspects of an acceptable solution to a problem. In this phase, we look at the computer and the people who need to use it. For example, a company may consider a number of ways of paying its employees: cash, computerized payroll checks, manually produced payroll checks or direct deposit to an account. [Ref. 25: p. 199] Other additional requirements must be considered before a selection of software or the development of software can begin: processing time, costs, error probability, chance of fraud or theft.

When designing a system, documentation should be designed first. Documentation is important in both the initial development of the system and in the subsequent maintenance. "A software specification and standard should require that the documentation to be provided on a project be specified. It should also be required that the various levels of documentation be consistent (e.g., sub-programs specifications should be consistent with the associated program specification)." [Ref. 26: p. 11] The following documentation can be found in varying degrees in computer software development projects in the phases indicated:

Functional Requirements Document	Problem Definition
Data Requirements Document	Problem Definition
System and Sub-System Specs	System Design
Program Specification	System Design

Data Base Specification	System Design
Test Plan	System Design
User Manual	Programming
Operations Manual	Programming
Program Maintenance Manual	Programming
Test Analysis Report	Test. [Ref. 27]

During the course of a software development project, oral communication and written documentation must be balanced for the best results of a project. "A requirements analysis can aid in understanding both the problem and the tradeoffs among conflicting constraints, thereby contributing to the best solution" [Ref. 25: p. 199]. Absolute necessities must be distinguished from bells and whistles. Time and space limitations, facilities plans for the future, and individual facilities requirements must be addressed. The money required for and the money available to implement the system must be considered. The management of the project must also be considered. As already discussed, PERT and CPM are popular methods of monitoring progress. "Once all these questions have been answered, specifications of a computer solution to the problem may begin" [Ref. 25: p. 199]. To summarize, what is needed is "a complete, validated specification of the required functions, interfaces, and performance for the software product" [Ref. 3: p. 37].

3. Preliminary Design/Product Design

When we look to determining the specifications of the software, we are actually asking what do we want the software to do? We want to determine , for example, the format of the input and output. What information would be desired for the production of a check and how should this information appear on the check. Algorithms must be considered for deductions from the basic check such as life insurance and health insurance plans.

A primary concern will be the size and content of the database. Beyond that, we will have to determine the layout of the database that will be most effective. If anything but a totally new system is being incorporated, plans must be made for conversion of the data in the old system to the new system. Compatability must be considered if new equipment is to be adopted to existing equipment.

The answers to these questions should be put forth in a document called functional specifications [Ref. 25: p. 199]. This document should be painstakingly prepared giving thorough definitions of the specifications required. The more complete this is, the fewer the errors will be in the final product. "Because it describes the scope of the solution, this document can be used for initial estimates of time, personnel, and other resources needed for the project. These specifications define only what the system is to do, but not how to do it." [Ref. 25: p. 199]

This theme of describing what and not how something is to be done is important for deriving the most from the programmers working on the project. If the how is to be defined by the person writing the specifications, he may be limiting himself to an antiquated solution to the problem and not availing himself of the creativity of the programmers. Herein we have once again an instance where a good manager will guide the development of the specifications and not unknowingly limit himself by doing the programmers job. With a basic knowledge of the system and programming, he will be able to clearly evaluate original solutions to the problems and employ the best technology available to the programmers.

4. Detailed Design

Much has been written about the design phase of a software development project.

> To reiterate, A complex system is one in which there are so many system states that it is difficult to understand how to organize the program logic so that all states will be handled correctly. The obvious technique to apply when confronting this type of situation is 'divide and rule.' This is an old idea in programming and is known as modularization. Modularization consists of dividing a program into subprograms (modules) which can be compiled separately, but which have connections with other modules. [Ref. 28: p. 66]

What is now considered to be the most effective way of developing a software project was set forth in a classic article by Stevens, Constantine and Meyers in 1974 and subsequently refined and developed by Parnas [Ref. 29], [Ref. 30], [Ref. 31].

Essentially, the concept of modularization is used. A particular design decision is assigned to one module. The job of coming up with an algorithm to implement that design decision is then given to one programmer or a group of programmers perhaps organized into programming teams as recommended by Brooks [Ref. 7: pp. 29-37]. When the work is modularized, it becomes easier for the programmers to understand. Communication lines can be established between programming teams so that questions can be answered. Each module is developed as an entity in itself and how it does its job becomes the secret of the module. The module will require certain inputs and will deliver certain outputs. The internal workings of that module will not be revealed to designers of other modules. The module will then not be tampered with.

> The connections between modules are the assumptions which the modules make about each other [Ref. 32]. Modules have connections in control via their entry and exit points; connections in data, explicitly via their

> arguments and values, and implicitly through data referenced by more than one module; and connections in the services which the modules provide for one another [Ref. 28: p. 66].

The beauty of this concept is that development time is shortened and modifications can be more easily made to one black box, the module, when changes are required down the line.

5. Code and Debug

During the code and debug phase, software is actually produced that meets the specifications and is certified to meet the user's requirements. Code is said to be verified when it meets the specifications of the design; code is said to be validated when it proves to do what the user wants it to do.

When converting data to code, errors are oftentimes made that are not easily detected. Wrong character usage can be caught without much trouble but correct characters used improperly will pass undetected.

> The credibility of data is often directly related to the origin of coding. Coding at the data source may lead to inadvertent errors due to a misunderstanding of the coding structure or carelessness in applying valid and relevant codes. Trained coders, selected and supervised with care and motivated as to the importance of their job, make fewer errors. [Ref. 23: p. 163]

6. Debugging and Testing

Since computers are not forgiving in nature and react to any errors, testing and debugging is extremely important. After each module has been coded, testing and debugging should be done; after each module has been tested separately, all the modules must be tested together as a system. System tests including acceptance testing are of course very important.

We can classify programming errors according to three types:

1. Syntax
2. Code Logic
3. Problem Logic

Syntax errors include such problems as omitted parentheses, incorrectly spelled (and thus unrecognizable) variable names, wrong data codes and miscounted character lengths. Compilers are used to find these errors.

Code logic errors are not as easy to find and include operable statements that produce incorrect results.

> Some such bugs are obvious-a misspelled word or misaligned title on an output report, for example. Other errors are difficult to discern, such as transferring control incorrectly after an IF statement and bypassing some intended instructions. Still others are insidious-for example, errantly substituting one variable name for another in an equation. The results may seem undecipherably random. [Ref. 33: p. 311]

Problem logic errors exist when the program does not adequately address the user's problem. For example, although a program may be correct for payroll, a wrong understanding of the tax laws or the payroll deductions by the programmer may render the output of the program useless to the user.

Historically, testing took a major share of the effort devoted to a project, often as much as 50%. With increased emphasis being put on the front-end development of a program, this phase is consuming less of the resources of the project and is generally consuming about 34% of the effort.

7. Operations and Maintenance

This phase concerns implementing the developed software in production and keeping that software functional. A number of areas are to be considered:

1. Operating personnel and computing facilities must of course be available
2. Errors that arise from usage must be corrected
3. Modifications must be made to the software as the user requirements change
4. Changes must be made as efficiency requirements change. [Ref. 34: P. 32]

B. ACTIVITY DEFINITIONS

Once the lifecycle phases have been defined one should estimate for each phase the fraction of the total amount of resources that are to be allocated to it. The activities to be performed in each of the phases should then be determined and resources assigned accordingly. [Ref. 35: pp. 625-631]

A typical allocation of resources in custom software development and test is:

1. Requirements Analysis: 8%
2. Preliminary Design: 18%
3. Interface Definition: 4%
4. Detailed Design: 16%
5. Code and Debug: 20%
6. Development Testing: 21%
7. Validation Testing and Operational Demonstration: 13%

Summing the four phases prior to code and debug shows that we allocate 46% of our total dollar there, 20% goes to coding, and the remaining 34% goes to the two major phases that follow coding. [Ref. 35: p. 630]

In order to enhance the reader's understanding of just how the dollars are being spent, a description of the activities involved is presented. A breakdown of the tasks performed within each activity during each phase is presented in Table I. The completion of each major phase of the software life cycle requires that various functions or activities be performed during each phase. We summarize these activities as follows:

1. Requirements Analysis: Determination, specification, review and update of software functional, performance, interface, and verification requirements.
2. Product Design: Determination, specification, review and update of hardware/software architecture, program design and data base design.
3. Programming: Detailed design, code, unit test, and integration of individual computer program components; includes programming personnel planning, tool acquisition, data base development, component level documentation, and intermediate level programming management.
4. Test Planning: Specification, review, and update of product test and acceptance test plans; acquisition of associated test drivers, test tools and test data.
5. Verification and Validation: Performance of independent requirements validation, design verification and validation, product test, and acceptance test; acquisition of requirements and design verification and validation tools.
6. Project Office Functions: Project level management functions; includes project level planning and control, contract and subcontract management, and customer interface.
7. Configuration Management and Quality Assurance: Configuration management includes product

TABLE I

Project Tasks by Activity and Phase

Activity	Plans and Requirements	Product Design	Programming	Integration and Test
Requirements analysis	Analyze existing system, determine user needs, integrate, document, and iterate requirements	Update requirements	Update requirements	Update requirements
Product design	Develop basic architecture; models, prototypes, risk analysis	Develop product design; models, prototypes, risk analysis	Update design	Update design
Programming	Top-level personnel and tools planning	Personnel planning, acquire tools, utilities	Detailed design, code and unit test, component documentation, integration planning	Integrate software, update components
Test planning	Acceptance test requirements, top-level test plans	Draft test plans, acquire test tools	Detailed test plans, acquire test tools	Detailed test plans, install test tools
Verification and validation	Validate requirements, acquire requirements, design V & V tools	V & V product design, acquire design V & V tools	V & V top portions of code, V & V design changes	Perform product test, acceptance test, V & V design changes
Project office functions	Project level management, project MIS planning, contracts, liaison, etc.	Project level management, status monitoring, contracts, liaison, etc.	Project level management, status monitoring, contracts, liaison, etc.	Project level management, status monitoring, contracts, liaison, etc.
CM/QA	CM/QA plans, procedures, acceptance plan, identify CM/QA tools	CM/QA of requirements, design; project standards, acquire CM/QA tools	CM/QA of requirements, design; code, operate library	CM/QA of requirements, design; code, operate library; monitor acceptance plan
Manuals	Outline portions of users' manual	Draft users', operators' manuals, outline maintenance manual	Full draft users' and operators' manuals	Final users', operators', and maintenance manuals

[Ref. 3: p. 50]

identification, change control, status accounting, operation of program support library, development and monitoring of end item acceptance plan; quality assurance includes development and monitoring of project standards, and technical audits of software products and processes.

8. Manuals: Development and update of users' manuals, operators' manuals, and maintenance manuals. [Ref. 3: pp. 46-50]

C. SUMMARY

A software development project's major phases and the activities of each phase have been presented. We feel that a manager needs a sound understanding of this aspect of software engineering economics if he is to not only understand but also contribute to his organization's development effort. The foundation of knowledge that is laid here concerning the software lifecycle (and as is true for all the ideas set forth in this research) will be built upon and refined as the organization interacts with professionals in the computer industry. With a sound, working knowledge of software engineering economics, managers will increasingly find that they are assisting in the development of an information system that fulfills their needs in an efficient and effective manner.

IV. EFFORT, TIME AND COST ESTIMATION

Herein we look specifically at the factors affecting effort, time and cost estimations. We feel that focusing our attention on this particular area of software engineering economics is essential for it is here that the organization's life-line is tapped. Effort, time and cost estimates will directly affect the stability and solvency of a company. Inaccurate estimates according to Murphy's Law will prove to be underestimates and accordingly drain the company of added resources that may or may not be conveniently available. A project may be scuttled due to the inability to provide additional support.

A. TIME AND EFFORT ESTIMATING

1. Experience and Judgement

Every estimate is influenced to some extent by the experience and judgement of its author. Some items influencing the estimate are so well understood that judgement seems to be replaced by the mere mechanical application of a rule, while others depend heavily upon the experience of the estimator. [Ref. 36: p. 48] The person responsible for ensuring the validity of an estimate should remain well aware of the skills and qualifications of the individual who prepared the estimate to give him/her a basis for determining its accuracy.

2. Programmer Productivity

Programmer productivity plays a major in part in estimations of the amount of time and effort that will be expended on a software development project. The paragraphs

that follow focus on some of the more important aspects of productivity. As productivity increases, software development costs decrease. In addition to worker quality and motivation, productivity depends on the use of advanced technology and the proper use and training of workers to effectively interface with the new technology. Short-term investment in training and job modification should lead to savings in the long-run due to increased productivity. [Ref. 37]

There are certain non-human elements that can have a great effect on productivity. The development environment is a key factor in this regard. One must ensure that adequate hardware and software support is available to the programmers. It is not uncommon for projects to become bottlenecked because throughput capacity, disk space, CPU capacity, or the like have been exceeded. The demand for these computing resources during design, development, integration and test is generally greater than during operations. The delays caused by such bottlenecks result in high levels of frustration and lower productivity among the programmers. [Ref. 38]

It should also be noted that poor programmer productivity is as much the result of bad management decisions and planning as it is the result of inadequate tools, or lack of talent. Productivity is affected by an organization's structure, goals, product type and experience in developing software. Care should be taken to ensure that an organization's software development process does not become a hindrance to productivity through imposed inflexible management procedures. [Ref. 39]

According to Jack Stone there are certain changes that could be made to the programmer's physical environment to increase his/her productivity. One of his suggestions is to give each programmer a private office to ensure quiet

surroundings rather than grouping the programmers together "like cattle in a box car". Another of his suggestions is to ensure that the programmer has available to him/her state of the art computer services (a CRT terminal with on-line interactive operating system controls, editors, compilers, and debug facilities). [Ref. 40] As previously discussed, improved programmer productivity is among the potential benefits that may be derived from the use of SDE.

3. Code Production Rates

A working standard of the typical code production rate per programmer man-month is 1 object instruction/ man-hour, which is equivalent to 156 instructions/man-month, or 1870 instructions/man-year for nontime-critical software. Wide variations in programmer productivity do exist however. [Ref. 35: p. 631]

4. Basic Manloading Pattern Over Time

Research on the man-effort loadings of medium to large scale software development projects has revealed a basic manloading pattern over time. Initially, there is a rise in man-effort followed by a peaking and then an exponential tailing off. The time varying nature of a project's work profile is to be expected since software development itself is a time dependent process. [Ref. 41: p.128]

> Consider the following rationale. A software project can be thought of as entailing the solution of a fixed number of problems. At each point in time, t, both the number of unsolved problems available for solution and the level of skill available for solving problems will vary. Since the rate of problem resolution is influenced by both factors, it too will be a time dependent process. Presumably, consumption of project resources reflects the rate of problem resolution, hence, the time varying nature of the manloading curve. [Ref. 41: p. 128]

B. COST ESTIMATING

1. Cost Considerations

A detailed understanding of the factors that impact on the cost of a software development project is required in estimating its cost. Two major problems are involved in the estimation of software development costs. One of these is the level of uncertainty and risk. The other problem is the lack of a quantitative historical cost data base. [Ref. 42: pp. 16-17]

Three factors contribute to the amount of risk and uncertainty involved. These are that the requirements are subject to change, something new may be required during the development process, and risks are inherent in the software development process itself. [Ref. 42: p. 17]

For a good software cost estimate one should work from firm requirements, understand the required product well, and carefully manage the development cycle to ensure that coding does not begin before the design has been thoroughly worked out, verified, and validated [Ref. 42: p. 17].

Without accurate measures of prior costs it is extremely difficult to estimate the cost of a new project. To solve this problem cost summaries should be archived and distributed by the project manager of the development effort to the appropriate personnel for estimation purposes. [Ref. 42: p. 17]

2. Key Factors Influencing Software Development Costs

The key factors influencing the cost of a software development project may be divided into the following four categories:

1. Requirement Factors
2. Product Factors

3. Process Factors

4. Resource Factors

a. Requirement Factors

(1). Quality of Specifications. Incomplete requirements definition is a major cause of cost overruns. The developer interprets the vague, poorly written requirements, prices the software package on the basis of that interpretation, and proceeds to design the software on that same basis. [Ref. 42: p. 17]

One of the keys to accurately costing software is to devote extra effort in solidifying the requirements before entering the detailed design phase of a project. Understanding the requirements is the basis for analysis of many of the other costing factors, including difficulty, interfaces, size, tools, use of existing software, and data base complexity. Poor estimates of software size or data base complexity are often blamed for cost overruns, when the actual reason for errors in these estimates is incomplete or inadequate specification of requirements at the outset of the initial software costing. [Ref. 42: p. 17]

(2). Stability of Requirements. There are many projects for which the well specified requirements against which the detailed design is prepared change during the project. It is the responsibility of the project manager to fully understand the software requirements and to ensure that it is understood that changes in the requirements baseline are just that, changes! The project manager should then define the cost and/or schedule impact so that the change may be fairly evaluated. If the change justifies the estimated impact on the project, a decision to incorporate it may then be made. The change should then be reflected in the requirements specification and incorporated into the

design; its impact on the project budget and schedule should also be stated. Once the impact of changes on the project is known, many changes that at first appeared attractive lose their appeal. [Ref. 42: p. 17]

b. Product Factors

Product factors are those factors derived from the characteristics of the software product to be developed and delivered, including both code and documentation. Following is a discussion of the six product factors.

(1). Software Size. A very common method of costing software is to estimate the number of instructions to be developed and multiply by a "magic number" (dollars per instruction) to get the estimated development cost. Although this estimating technique is not very precise when used alone, it can be very useful when used in conjunction with the other factors.

Significant sizing considerations include the following:

1. Care must be taken to isolate the deliverable software from the nondeliverable test software, simulations, and support software, which should be less costly to produce.
2. As the size of the software increases, other factors such as complexity, interfaces, and the number of people involved, begin to have a greater influence on the cost.
3. When trying to use size as a costing parameter, care must be taken that the cost base being used is derived from the same sizing parameter. Projects or companies may track costs by lines of code, number of object instructions, number of executable source statements, total instructions or lines of code developed, or delivered instructions or lines of code.

4. When using size/cost factors, consideration should also be given to productivity differences between languages.
5. When object code sizing estimates are based on similar existing software, consideration should be given to differences in the expansion ratio from source to object instructions between different HCL's, compilers for the same HOL, or different operating systems.
6. As size increases, the number of individuals involved in the development effort increases and the amount of time spent in intercommunication and coordination becomes significant, driving the cost versus size from linear toward some higher multiple. [Ref. 42: pp. 20-21]

(2). Difficulty. One of the more important factors affecting software development costs is the relative difficulty of the software application. Software personnel productivity (and therefore cost) will vary with the type of system being developed. Real-time applications are generally considered to cost up to five times as much as HOL nonreal-time applications. [Ref. 42: p. 21]

(3). Reliability Requirements. According to Bruce and Pederson the reliability of a software program may be determined by four major criteria. These are:

1. the program must provide for continuity of operation under nonnominal circumstances;
2. the design, implementation techniques, and notation utilized must be uniform;
3. it must yield the required precision in calculations and outputs; and
4. the program must be implemented in a manner that is understandable.

As the level of requirements for handling nonnominal conditions increases so does the amount of verification effort required and, along with it, the cost. [Ref. 42: p. 21]

(4). External Interfaces. Cost increases as the complexity of external interfaces increases due to the additional effort required for design, implementation, and integration [Ref. 42: p.21].

(5). Language Requirements. Experience has shown that it takes an average programmer about the same amount of effort to write a line of code in high order language as in an assembly language. Apparently the thought process required to write a single statement is almost independent of the language in which the statement is written. It will take a programmer significantly longer to write a program in assembly language than it would to write the same program in HOL, since a typical HOL statement expands to 5-10 assembly language statements. Early in a project a programmer's familiarity with a language will affect the cost per statement more than the language being used. [Ref. 42: p. 21]

(6). Documentation Requirements. The cost factors associated with the preparation and acceptance of required documentation must be evaluated along with all other cost factors [Ref. 42: p. 21].

c. Process Factors

Management structure, management controls, tools, use of available software, and data base methods are all software costing factors associated with the development process. A discussion of these factors follows. [Ref. 42: p. 21]

(1). Management Structure. Management structure effects the organization's policies regarding the allocation of resources for a software development project [Ref. 42: p. 22]. If the structure is such that upper level management arbitrarily imposes standards without understanding their purpose, use, or implications on the software development process, the standards may prove to be counterproductive. Management should tie software development to organizational and product goals and ensure that the process is usable at the working level. [Ref. 39] The structure should be such that the programmers and engineers are able to get what they need when they need it without the hassle of having to get requests through an inflexible approval chain.

(2). Management Controls. This factor covers the cost of project support in such areas as management information processing, scheduling support, and clerical support. The cost estimator must realize the need for this type of support and have some understanding of the relative magnitude of this type of project cost. [Ref. 42: p. 22]

(3). Development Methods. This factor attempts to quantify the impact of various development methods. The development methods of interest include such approaches as top-down design and testing, structured programming, use of chief programmer teams, and use of structured walk-throughs. [Ref. 42: p. 22]

(4). Tools. The cost estimator must consider how the software will be developed, tested, and maintained and what tools will be needed to accomplish these tasks. For some projects the development of software and hardware tools is a major cost item. The cost estimator must determine whether compilers and other tools are required, available, need to be converted, or need to be developed. The costs associated with the tools are a function of the

tool complexity, use, features, and maturity. Experience provides the best basis for analyzing the cost impact of support software and tools on overall project cost. [Ref. 42: p. 22]

(5). Available Software. Significant reductions in the cost of projects may be achieved through the use of existing software. Adapting the existing software as part of a system requires analysis of the software apart from the new development. The costs for modifying the existing software can in this way be determined subjectively. Care must be taken to include the cost of interfacing the modified software to the new software and revalidating the requirements. [Ref. 42: pp. 22-23]

(6). Data Base. The size, complexity, and special file access requirements for the data base are very important parameters in deriving an accurate software development estimate. The cost estimator must review the data base requirements and subjectively analyze their impact on cost. [Ref. 42: p. 23]

d. Resource Factors

Software development costs for a given project may vary substantially, depending on such factors as the experience of the available personnel, the quality of the project staff, and availability of development computer resources [Ref. 42: p. 22].

(1). Number of People. With projects that require large staffs the major contributor to the reduction in productivity (increased cost) is the increase in the time needed for communication between the people [Ref. 42: p. 23].

(2). Experience of People. Existing data indicates that there is no direct correlation between the number of years of experience that a person has and his/her

productivity. However, experience with a specific type of application does have an effect on the development effort required. Generally speaking, a programming group will require from 50-100% more effort to develop a variant of a previously developed, familiar program. [Ref. 42: p. 23]

(3). Personnel Performance. Individual productivity variations are to be expected in the development of software due to the fact that it is an analytical, and sometimes creative activity that requires abstract reasoning. Nonetheless, experienced estimators have found variations in productivity to be as high as 10:1. The assessment of productivity is extremely important because cost estimation is generally reduced to deriving a productivity figure per unit of effort per person within a given skill category. The use of such average productivity figures for estimating cost tends to even out for large projects, but may prove to be disastrous for small projects. [Ref. 42: p. 23]

(4). Availability of Computing Resources. As the requirement for computer time increases during the development cycle, the impact of insufficient computing resources on schedule and cost increases. The amount of computer time required for a given development effort is easily underestimated. [Ref. 42: p. 23]

(5). Suitability of Computing Resources. During the software maintenance phase, when there may be little capacity available for corrections, modifications, or required test drivers to verify changes, there is an asymptotic effect on development costs as the hardware speed and memory size constraints are approached which could prove to be crippling [Ref. 42: pp. 23-24]. A normal person would not ordinarily jump into a sports car and speed off down some winding mountain road he had never driven on before in the black of night. If he did, without warning, he could find himself at the bottom of a canyon, surrounded by steep cliffs.

3. Traditional Cost Estimating Procedures

Traditional cost estimating procedures begin by fixing the size of each activity and determining its start date and duration. When and if it becomes necessary to do so, adjustments are made to account for the skill levels of the assigned personnel, the complexity of the project, and the degree of uncertainty in the requirements. The amount and type of manpower and resources are then converted to dollar costs. Other direct costs, such as documentation and travel, are also added in.

Traditional cost estimating methods include:

1. Top-Down Estimating: The estimate obtained using this method is based on the total cost or the cost of large portions of completed projects. A problem with this method is that it carries with it the risk of overlooking important technical problems that may not be readily apparent. [Ref. 35: p. 618]
2. Similarities and Differences Estimating: In this method jobs are broken down to a level of detail where the similarities to and differences from previous projects are most easily recognizable. Those units that cannot be compared to previous projects must be estimated by some other means. [Ref. 35: p. 618]
3. Ratio Estimating:

> The estimator relies on sensitivity coefficients or exchange ratios that are invariant (within limits) to the details of design. The software analyst estimates the size of a module by its number of object instructions, classifies it by type, and evaluates its relative complexity. An appropriate cost matrix is constructed from a cost data base in terms of cost per instruction, for that type of software, at that relative complexity level. [Ref. 35: pp. 618-619]

The appeal of using this method is in its simplicity, speed, convenience, and usefulness in a variety of environments. A major shortcoming is in the lack of a valid cost data base that covers a number of estimating situations. [Ref. 35: p. 619]

4. Standards Estimating: In standards estimating, systematically developed standards of performance are depended upon. New tasks are calibrated from these standards. This method is reliable only for repeatedly performed operations that have been well documented. The rub is that the same software development projects are not performed over and over again. [Ref. 35: p. 619]
5. Bottom-Up Estimating: Government research and development contracts are most generally estimated using the bottom-up approach. A work break down is done on the project until it is reasonably obvious what steps and resources are required for each task. The costs are then estimated for each task and a pyramid is developed to estimate the total cost for the project. Using this technique, the estimating assignment can be given to the people actually doing the work. One problem with this technique is the inavailability of the total cost structure at the inception of the cost estimating job. [Ref. 35: p. 619]

4. Cost Estimating Relationships and Phase Interrelationships

Software cost estimations should include the effects of resources consumed in one lifecycle phase on subsequent phases. A large contribution to the resource requirements for any one phase derives from the ways in which the other phases are completed. An important factor affecting the utilization of resources is the need to conform to a

development plan. The plan is an essential management tool for ensuring that the needed resources are available for the project at the right time and in the correct amount. Changes in the plan, whether caused by changes in requirements or by failure to meet commitments, may affect cost-driving parameters. [Ref. 43: p. 70]

V. THE ART AND SCIENCE OF SOFTWARE COST ESTIMATION

Until absolutely reliable, comprehensive methods of estimating cost and effort in software development projects are developed, the techniques will be referred to both as an art and a science. We appropriately use the term science as estimating techniques are becoming more and more accurate and comprehensive. Mathematical and scientific principles are increasingly being applied to all areas of cost and effort estimation. Researchers are now developing models that can be used in numerous environments.

A. CURRENTLY AVAILABLE METHODS FOR SOFTWARE COSTING

Many models estimating cost and effort exist on the market today and generally cover the time from the design and specifications phase thru the test and debug phase and the beginning of operations. They can ordinarily be classified as theoretical or empirical. Theoretical models are those based on global assumptions such as the rate at which people solve problems. Empirical models use information from former projects to evaluate current projects and derive basic formulas from the available information in the data base. [Ref. 3: p. 511] We will present a number of available cost and effort estimating models according to their classification as static, dynamic or dynamic transportable models. We will examine some models in more detail than others to give the reader added insight into the complexity of estimating cost and effort. Some of the more significant features of the models will be pointed out. We will then enumerate criteria which may be used to judge a model for estimating cost and effort in software development projects.

1. Static Models

Models that do not treat time explicitly and do not have the capability to adapt to the actual behavior of the system at any instant of time during the lifecycle are termed static models.

a. Doty

The Doty model estimates the manpower, cost and development time for software development projects. System size is estimated by comparisons of the system under consideration to comparable known systems. The model is therefore empirical. Doty found that the writing of high order languages (HOL) and assembly language instructions takes the same amount of time. Since HOL programs are smaller than assembly language programs, productivity is increased with HOL programs. Clarity and maintainability are higher with HOL. [Ref. 2: p. 108]

b. "SOFCOST"

In recognition of the need to establish good cost estimations before proceeding on a software development project, Grumman Aerospace Corporation has developed an empirical model to provide viable, credible cost estimates. Before completing its own model, Grumman used the Price-S model to estimate costs. Presently, both the "SOFCOST" model and the Price-S model are used in parallel as independent cost estimates to act as checks and balances for estimates of the project system analyst. "SOFCOST" allows the analyst to estimate the effort and elapsed time to complete a software development project. It is a parametric model developed from statistical software history. This empirical model uses for its primary parameter functional size. The basic estimating relationship is influenced by:

1. Experience
2. Complexity
3. Environment
4. Technology
5. Hardware
6. Reliability
7. Language
8. Requirement Definition.

The model operates interactively with the user to develop a software work breakdown structure (SWBS), a functional size matrix for the SWBS and the time and effort computed for each item in the SWBS.

> There are five levels to the SWBS, the computer program configuration item, the category of software, the functions per category and two output levels - task and phase. The two output levels provide the manpower tasks of technical, support, management, configuration control and documentation per development phase of definition, design, code, test, intergration and acceptance for each function in the SWBS. The number of system computer resource hours is also computed and provided as output.
>
> "SOFCOST" also derives an elapsed time schedule for each of the functions in the SWBS providing durations for each of the phases included in Level Five of the SWBS. A cumulative schedule is computed providing for overlap in each phase. [Ref. 44: p. 674]

Grumman's research included 30 different models and a review of research conducted by industry and government. The work resulted in a requirements and design document for an in-house model. The model not only included prior software/cost relationships but also charcteristics unique to the Grumman environment.

Research concluded that the primary cost driver is executable lines of source code. "SOFCOST" was therefore designed to aid the user in estimating the number of lines of code. The estimator can make comparisons between his function and comparable functions found in the data base including function and size as its key parameters. The

determination of size of a project is thus a critical factor and one that is addressed using the judgement of the analyst.

"SOFCOST" has three objectives:

1. to construct an SWBS,
2. to determine a credible size for the functions being estimated,
3. to estimate software cost and schedule for each functional task. [Ref. 44: p. 674]

As is becoming increasingly common in literature on the topic, Grumman feels that the interactive user developing an estimate for cost and effort for a project should be knowledgeable in computer software design and the particular system's requirements. The SWBS will be established in an interactive session. After the five levels are completed, the user interacts with the program to answer questions that affect the basic cost computation. Costs are given in manhours or manmonths and elapsed time schedules are displayed.

Translator 1 establishes the SWBS. Translator 2 establishes a size estimate for all functions in the SWBS using functions of a comparable nature from the database. Translator 3 of the program takes the output from translator 2 and computes manpower effort and schedules elapsed time. It is here that the estimator begins to interact to determine the adjustments to the basic estimates. Language is first considered. Prior studies showed little difference between productivity of HOL. Differences occurred in the productivities of HOL's compared to assembly language in the order of 2 or 3 to 1 improvement in productivity (this is in keeping with Doty's findings).

> After the language type is established in Translator 3 and an adjustment made in the size of the source code for language the basic manpower versus line of code for relationship in the model is exercised. "SOFCOST" computes the effort for the phases of design, code and test with a parametric equation for each phase. The effort computed during the design phase includes those steps in the software development cycle of requirement definition, preliminary design and detail design. The code phase equation output includes coding, debugging and module test. The test phase effort includes subsystem, system, integration, and acceptance testing. The equations for the design, code and test efforts were regressed from historical data published from studies conducted by SDC, GRC, IBM and TRW and from actual data produced at Grumman. These regressions when taken with various combinations of the published source data produced correlation coefficients in excess of 85% when converted to the log-linear form. The F value measure of statistical acceptability based on the number of observations in the regression were on the average greater than 200 and indicative of regression significance. [Ref. 44: p. 677]

Interactions are then performed to adjust the basic computation effort. Thirty questions are asked the user and he evaluates each on a scale of 0 to 10. The inputs are used to derive a productivity index factor. Adjustment factors are computed for each question. Individual adjustments are weighted. Table II lists the adjustment factors in weighted order.

An adjusted manpower effort is computed after all of the individual adjustment questions have been answered.

> This adjusted effort is then distributed among the phases of definition, design, development, test, integration and acceptance in accord with the results of published history. This is a variation of the standard 40-20-40 allocation. This adjusted computed effort represents the technical effort expended upon the embedded program (application, tactical) by the personnel assigned as programmers, analysts, systems engineers, etc. Translator 3 then takes this computed effort and determines the support, management, and configuration/quality control efforts as some function of the technical effort. Both the documentation and computer resource efforts are computed separately using a parametric equation relationship formulated for these tasks. [Ref. 44: p. 678]

TABLE II

Adjustment Variables by Decreasing Weight

Percent of real time programming design
Percent of new algorithm design
Percent of existing code reutilized
Percent of requirements ill-defined
Percent of time share facilities employed
Percent of pushing the state of the art
Number of interfacing displays
Number of interfacing equip excluding displays
Percent of user experience
Percent of concurrent hardware/software development
Percent of code inspection technique employed
Percent of change anticipated for the program
Percent of top down design employed
Percent of computer time utilized as a design goal
Percent of security in design
Percent of structured programming employed
Percent of input/output control programming design
Number of average years experience of personnel
Percent of previous experience with the computer
Percent of application/functional experience
Percent of chief programmer team technique employed
Percent of memory capacity utilized as a design goal
Number of programming locations
Percent of software personnel experience
Number of instructions in the computer set
Percent of language experience
Percent of previous experience with similar algorithms
Percent of user defined requirements alone
Length of the computer instruction word
Percent of user/contractor interface complexity
[Ref. 44: p. 677]

For scheduling, elapsed time is computed both as a function of the computed manpower effort and also as a function of the adjusted lines of code. Start and end dates are computed for each phase. An optimized schedule is output and differences between planned schedules and optimized schedules are highlighted. Requirements documents usually dictate planned schedules and recognition of accelerations and/or stretchouts that might happen if the planned/contract schedule were followed.

Thus, "SOFCOST" uses historical data and empirical data from the environment to develop estimates of cost in manhours or manmonths and scheduling for various phases of a project. The interactive sessions with the user allow a more clearly defined SWBS. The primary cost driver was found to be executable lines of code. The basic computations are adjusted by an interactive session with the user in which specific environmental factors are evaluated and accounted for.

The key factors in this model that are critical to the estimation effort are the initial sizing of the project and the determination of unique environmental factors that affect costing and scheduling. In both of these activities, the judgement of the estimator as he interacts with the computer is critical to the success of the project.

c. Lifecycle Cost Estimating

The bottom-up decomposition methodology and a top-down regression analysis is used at the conceptual requirements level to provide fast and accurate estimates of software lifecycle costing (LCC) employing unique software structures. The software structural model is further analyzed and manipulated to give useful design alternatives in the form of such criteria as program control, logic paths, and data transfer that improve the operational qualities of the software and provide minimum LCC designs. This technique seeks to obtain a uniquely realizable decomposition strategy and finally give a machine designed cost-effective software structure. Silver feels that the current method of using an empirical top-down approach and multiple regression analysis employing extensive data bases to estimate software sizing and costing is unsatisfactory. The uniqueness of each software package precludes this approach.

> The overall problems of program management and cost control, as well as the selection of cost-effective design alternatives are addressed by using a combined Graph Theory "bottoms-up" decomposition methodology to provide accurate and rapid assessments of both technological feasibility and economic risks in conjunction with a "tops-down" regression analysis employing cost estimating relationships (CERs). At the software requirements and conceptual level, structural decomposition identifies critical milestones and exposes subsequent cost drivers through the specification of connectivities and paths which yield minimum Life Cycle Costs (LCC). This is accomplished by utilizing the properties of agglomerative polythetic clustering to define a topology for determining objective decomposition strategies related to computer software structures. The mathematical basis for mapping the software structure onto a particular graph metric space is discussed in terms of formulating a quality index for operational structural partitioning. The use of potential multi-attribute semi-metrics is illustrated with a view towards obtaining an optimal, decomposition strategy and ultimately provide machine designed cost-effective software structures. [Ref. 45: p. 665]

Silver found that the traditional methods fail to provide a comprehensive and useful software management equation that has terms that are functionally separable and independently linearly related to system qualities such as file structure, memory requirements, and number of application programs. The equations are all too often complex. The subjective role of the estimator is not accounted for. Silver feels that the top-down approach does not give the necessary accuracy and certainty of estimations to be exhaustively useful in estimating cost and effort in software development projects. A methodology should give accuracy and certainty early in the development process of the cost and effort involved.

"The assumption is inherently made that attributes of design quality at the requirements level are sufficiently manifest in the structural characteristics of the design process itself, so that they can be costed in detail. Furthermore, the analysis of a given software structure is an appropriate vehicle for comparing different strategies on a cost-effective basis." [Ref. 45: p. 667]

Attempts have been made to explain a methodology for the characterization of the software design process in terms of a software structural framework.

> The essential conclusion reached is that software structural decompositions may indeed serve as the basic underpinning for the design activity and associated cost/performance specifications at the requirements level. [Ref. 45: p. 667]

Our look at this method is concluded with the author's remarks:

> The conclusions emanating from this study will be deferred to a more comprehensive paper dealing with the details of the methodology. The intent of this investigation is to lay the foundation for decomposition and recombination without resorting to excessive rigor, while at the same time report some interesting results. [Ref. 45: p. 671]

d. GRC

Cost is figured as the non-linear function of the number of delivered instructions. This model has numerous different estimating relationships which are difficult to summarize. It has a number of good features, including a thorough definition of the quantities being estimated and a set of relationships for estimating such quantities as training and installation costs and labor-grade distributions. Some drawbacks, however, include the use of 'number of outputs formats' as the basic size parameter and some evident typos or mistakes in the 0.0 values given in the effort multiplier tables. [Ref. 3: p. 519] System development cost is generally reduced if excess processor capacity is available, especially for virtual memory systems. The model considers the maximum processor capacity utilized in estimating the constrained software cost. [Ref. 2: p. 105]

e. TRW

The empirical TRW-Wolverton Model assumes that the total effort exerted in completing a program is linearly proportional to the number of instructions to be produced. The following is used in this model: cost-per-instruction matrix, organized by software category (control, I/O, pre-processor/post-processor, algorithm, data management, and time-critical) and degree of difficulty (old program- easy, medium, hard; new program-easy, medium, hard). An historical computer usage matrix is kept by category of software to estimate the cost of computer time needed for a project. The net cost becomes a product of cost per instruction and the projected number of instructions to be produced. Wolverton has noted in his analysis that past experience does not impact on programmer productivity significantly. [Ref. 2: p. 104] The heart of the estimate is a number of curves showing software cost per object instruction as a function of the relative degree of difficulty (0-100), novelty of application, and type of project [Ref. 3: p. 512]. Software is broken into parts and costs estimated individually in the best use of the model. "This model is well-calibrated to a class of near-real-time government command and control projects, but is less accurate for some other classes of projects. In addition, the model provides a good breakdown of project effort by phase and activity." [Ref. 3: p. 513]

2. Dynamic Models

These models use real time input and indicate where we are now and where we are going at a particular instant in time.

a. TRW (SCEP)

The new TRW Software Cost Estimating Program (SCEP) was developed by Boehm and Wolverton. This model was developed using the set of criteria presented later to evaluate software cost estimating models. Comments on this model's performance according to the criteria set forth will be given later in the overall analysis of the models.

b. Walston and Felix Model

This model gives a method for estimating programmer productivity. Programmer productivity is measured in the rate of production of lines of code (LOC). Given an estimate of the lines of code to be produced, the model estimates the total man-months of effort required. Man-months become a function of the LOC to be produced. From the data base of IBM-Federal Systems Division (IBM-FSD) consisting of 60 projects [Ref. 3: p. 406], a set of relations was developed to be used for cost estimation processes. The relationships are:

1. productivity vs percentage of new code
2. productivity vs percentage of effort at primary location
3. productivity vs percentage RJE use
4. delivered source documentation vs delivered code
5. duration vs delivered code
6. duration vs total man-month effort
7. staff size vs total effort
8. computer cost vs delivered code
9. computer cost vs total man-months of effort

The main problem with the model is the difficulty in determining how change in the ratings of productivity of cost drivers is due to other correlated factors or by double counting using four factors to account

for the use of modern programming practices [Ref. 3: p. 517].

c. Aron Model

Aron found that large system building efforts increase gradually, reach a peak, and then decline to zero. The peak time and system testing seem to coincide. He investigated the following ways of estimating software costs: experience, constraint, units-of-work, and quantitative. Experience depends on exposure to similar jobs in similar environments. Using constraints, the manager just agrees to do a job within given constraints. In the units-of-work approach, the job is broken down into smaller units, cost is estimated for each unit based on past experience with units of the same size. When quantitative estimation is used, the job is broken down into smaller tasks, classified as easy, medium or hard depending on interactions with other tasks. The man-months for each method is given by the deliverable instructions divided by the productivity, and total man-months is the sum of man-months for each task. [Ref. 2: p. 106]

d. Putnam Model

Empirical observations by Aron provided the basis for the Putnam model. Norden found that research and development projects reflected overlapping phases and he indicated them by the Raleigh form. Norden found that the work cycles of the Raleigh form have the characteristic of 90% of the work being done in two-thirds of the time with 10% of the work taking one-third of the time at the end. This gives the reason for the long delays at the end of a project. Putnam found that software projects usually conform to Rayleigh-Norden forms. "He related the system attributes, number of files, modules, and reports to the

manpower, understanding exactly what the software development process consists of over its life cycle, maintaining a data base that reflects the history of actual software development costs, and developing the most cost-effective allocation of resources to different phases of software." [Ref. 2: p. 106]

Putnam has developed Monte-Carlo simulation and linear programming to estimate development time and manpower from the trade-off law in the systems definition phase. "Other parameters that can be estimated are the contract milestones from computed development time, the impact of requirement changes during the development phase, optimal future resource allocation during the development phase, and computer usage and resource allocation during the operations and maintenance phase." [Ref. 2: p. 106] The SLIM model, the updated version of the Putnam model, also has the abilities of estimating computer costs and using the PERT sizing techniques.

3. Dynamic Transportable Models

Models that use real time information and are portable to different environments are termed herein dynamic transportable models. These models can be evolved to reflect specific environmental influences.

a. Meta Model

The Meta model is an empirical model based primarily on the work of Boehm and Walston & Felix. This model permits the development of a resource estimation model for any particular organization. The model itself can be used from the beginning of the design phase through acceptance testing and includes programming, management and support hours. Effort is expressed as some measure of size. Deviations from the average are explained by environmental

attributes known for each project. A background equation is computed, environmental factors analyzed and the model predicts effort for the project. A size measure is chosen from available data. Estimating size for each project is accomplished by taking the total number of new lines written and adding them to 20% of any old lines used in the project. A base-line relationship of lower standard error is derived. The size measure is called developed lines. Developed modules is arrived at in the same manner. Effort is measured in man-months. The Meta model is employed as follows:

1. Compute the background equation
2. Analyze the factor available to explain the difference between actual effort and effort as predicted by the background equation
3. Use this model to predict the effort for the new project. [Ref. 46: p. 108]

Collecting data about the environment is done as follows:

1. Choosing a set of factors
2. Grouping and compressing this data
3. Isolating the important factors
4. Incorporating the factors by performing a multiple regression to predict the deviations of the points from the computed base-line. [Ref. 46: p. 111]

As a rule of thumb, 10% to 15% of the number of data points should be the number of environmental factors used to predict a given number of points. The Meta model collects data from a particular environment and uses that data to make predictions about the environment.

Good managers can usually estimate the cost and effort of a software development project better than the predictions of a model brought in from another environment. The expectation is that this model will assist those managers in making even better predictions concerning cost and effort. The Meta model is developed by duplicating the basic steps of the model with information from a unique environment. The model is molded into the environment which will use it and not simply tuned to accommodate the new environment. The model itself is based on earlier works by Walston & Felix and Boehm who attempted to relate project size to effort. Measures used to express size in the Meta model are:

1. Lines of Source Code (LOSC)
2. Executable Statements
3. Machine Instructions
4. Number of Modules

A base line equation is used in conjunction with individual attributes of a project that affect the base line equation. Boehm and Walston & Felix have suggested similar models. Environmental differences explain variations from the averages arrived at by various equations. Environmental differences are accounted for by a number of factors such as:

1. Skill and experience of the programming team
2. Use of good programming practices
3. Difficulty of the project (complexity)

A two step approach is used to develop the model.

1. Effort exerted on an average project is expressed as a function of size.
2. Deviations from the average are attributed to environmental characteristics. The background equation is derived from the relationship between effort and

size. The measurement of size depends on the data available.

The use of the model is as follows:

1. Estimate size of new project
2. Use base-line to get standard effort
3. Estimate necessary factor values
4. Compute difference this project should exhibit
5. Apply that difference to standard effort.

[Ref. 46: p. 114]

The main difficulty with the Meta model involves identifying significant environmental factors and deciding how many to use in the estimating process. Tables III, IV and V include environmental factors identified by Walston & Felix, Boehm and those identified at the Software Engineering Laboratory at the NASA/Goddard Space Flight Center where Bailey and Basili collected the data to demonstrate the Meta model. For any particular project, attributes selected for study depend on what information is available in the data base.

Of the original 71 attributes that the researchers thought to have influence on the effort for a Meta project, 21 were selected for analysis and grouped into three major categories. Predicting a variable with a few data points (18) using many factors is not statistically sound. The problem with adding the points of each attribute that indicated its influence and using the sum for the influence of that category is that some individual factors that may be very influential lose their identity. Two ways around this are to use more data points and evaluate each attribute independently or to determine the relative affect of each attribute and weigh them independently. Bailey and

TABLE III

Evaluation Factors - SEL

Program design language (development and design)
Formal design review
Tree charts
Design formalisms
Design/decision notes
Walk-through: design
Walk-through: code
Code reading
Top-down design
Top-down code
Structured code
Librarian
Chief Programmer Teams
Formal Training
Formal test plans
Unit development folders
Formal documentation
Heavy management involvement and control
Iterative enhancement
Individual decisions
Timely specs and no changes
Team size
On schedule
ISO development
Overall
Reusable code
Percent programmer effort
Percent management effort
Amount documentation
Staff size
[Ref. 46: p. 112]

Basili did not have the requisite criteria for either solution so they used the described method of grouping.

b. Price-S and Price-SL

The Price-S and the Price-SL are empirical models developed by RCA and can be used in conjunction to estimate the software costs during a support period for a given project [Ref. 47: pp. 663-664]. The Price-S model uses a top down approach to determine the resources required in a software development project. The model delivers cost and schedule for size, type and difficulty of the subject

TABLE IV

Evaluation Factors - Walston and Felix

Customer experience
Customer participation in definition
Customer interface complexity
Development location
Percent programmers in design
Programmer qualifications
Programmer experience with machine
Programmer experience with language
Programmer experience with application
Worked together on same type of problem
Customer originated program design changes
Hardware under development
Development environment closed
Development environment open with request
Development environment open
Development environment RJE
Development environment TSO
Percent code structured
Percent code used code review
Percent code used top-down
Percent code by chief-programmer teams
Complexity of application processing
Complexity of program flow
Complexity of internal communication
Complexity of external communication
Complexity of data-base structure
Percent code non-math and I/O
Percent code math and computational
Percent code CPU and I/O control
Percent code fallback and recovery
Percent code other
Proportion code real time of interactive
Design constraints: main storage
Design constraints: timing
Design constraints: I/O capability
Unclassified
[Ref. 46: p. 112]

project. The Price-S model uses information from historical data bases to estimate the costs of a new project. The Price-S model gives information about the software when it is installed for operation. The Price-SL model uses information about the environment to estimate the cost to be incurred during a particular support period. Combining these two models, we arrive at cost estimates up to a particular point in the development phase and throughout a given support period.

TABLE V

Environmental Factors - Boehm

Required fault freedom
Data base size
Product complexity
Adaptation from existing software
Execution time constraint
Main storage constraint
Virtual machine volatility
Computer response time
Analyst capability
Applications experience
Programmer Capability
Virtual machine experience
Programming language experience
Modern programming practices
Use of software tools
Required development Schedule
[Ref. 46: p. 112]

The Price-S model provides the following cost drivers:

1. Instructions
2. Application
3. Platform
4. Development Schedule

Software size is measured in the number of instructions. Application refers to the type of software being developed. Platform refers to the environment in which the software operates. Development schedule is self explanatory. A development schedule is computed and compared with a design schedule and the degree to which the design schedule is normal, accelerated or stretched out will affect the amount of repair activity. Accelerated schedules will be more costly and stretched out schedules will cost less due to the extra time to develop better quality software.

The Price-SL identifies two primary cost drivers:

1. Support Schedule (SSTART to SEND)
2. Growth Factor

Shorter schedules will see bugs more quickly found but a lower total number of bugs. Shorter schedules will preclude enhancements to the system and the anticipated growth factor will probably be lower for short schedules. The number of installations and the amount of average usages will affect the number of bugs found. The higher either of these, the more bugs. Other support economic parameters are modifiers of the calculated costs and include multipliers for support mark-ups and support escalation.

Costs for the Price-SL are categorized as follows:

1. Maintenance
2. Enhancement
3. Growth

Software costs are estimated for the following five elements in each category:

1. Systems Engineering: technical tasks of the entire software system such as updating test plans and test specifications.
2. Programming: cost for implementing design and code changes.
3. Configuration Control: cost of maintaining system integrity and determination of system baseline.
4. Quality Assurance: cost of maintaining system integrity and determination of system baseline.
5. Documentation: cost of all changes needed to support Maintenance, Enhancement and Growth.
6. Program Management

Costs on a yearly basis are provided for the three major areas or the five elements. The Price-S and the Price-SL

models are available from RCA and can be used to estimate cost in varying environments.

c. COCOMO

The COnstructive COst MOdel detailed by Boehm in his most recent publication is a most powerful instrument for estimating cost and effort in software develcpment projects. The more detail that is provided as input to a cost estimation model, the more accurate the estimates will probably be. The COCOMO model allows the preparation of estimates in good detail and specifies and processes them with considerable efficiency. The following factors impact cost:

1. Cost Driver: Product Attributes
   a) RELY: Required software reliability
      i) Does the software perform its intended functions over the next utilization and subsequent utilizations?
      ii) DATA: Data base size
      iii) CPLX: Software product complexity
2. Cost Driver: Computer Attributes
   a) TIME: Execution time constraint
   b) STOR: Main storage constraint
   c) VIRT: Virtual machine volatility
   d) TURN: Computer turnaround time
3. Cost Driver: Personnel Attributes
   a) ACAP: Analyst capability
   b) PCAP: Programmer capability
   c) VEXP: Virtual machine experience
   d) LEXP: Language experience
4. Cost Driver: Project Attributes
   a) MODP: Use of modern programming practices
   b) TOOL: Use of software tools
   c) SCED: Development schedule constraint

Hierarchical decomposition is used to aid in producing cost estimates. The lowest level is the module. Cost drivers that are described at this level are: complexity and adaptation from existing software; programmer's capability level and experience with the language and virtual machine on which the software is to be built. The second level is the subsystem level. A number of cost drivers affect this level. The cost drivers vary from subsystem to subsystem but are usually the same for all modules in the particular subsystem. The top level is the system level. This level is used to apply overall project relations like nominal effort and schedule equations and to apply the nominal project effort and schedule breakdowns by phase.

For each cost driver, a set of tables is used to account for its affect on each major development phase.

4. Overall Model Evaluation

Boehm has enumerated a number of criteria upon which software cost estimating models can be evaluated.

1. Definition: do we understand from the model what costs it is estimating and what costs it is excluding?
2. Fidelity: do estimated costs compare favorably with actual costs?
3. Objectivity: are cost drivers related to factors that are objectively measurable and not open to manipulation to get what we want?
4. Constructiveness: is it clear from the model why a particular estimate is arrived at and is the software project more understandable because of the model?
5. Detail: does the model sufficiently breakdown the project for estimation purposes?

6. Stability: do small input changes produce small output changes?
7. Scope: is the model applicable to the type of project needed to be estimated?
8. Ease of Use: are the inputs and options used by the model easy to understand and specify?
9. Prospectiveness: does the model only use information that can be found before completion of the project? This criterion is used only for cost prediction.
10. Parsimony: are redundant factors and factors that do not contribute to the result of the model avoided? [Ref. 3: p. 476]

We will examine the models presented with respect to the applicability of a number of the above criteria.

a. Definition

The IBM-FSD model, the Bailey-Basili model and the 1979 GRC model provide fairly thorough definitions of the inputs and outputs used. COCOMO provides as thorough as possible definition of the activities and quantities found in the model while not overly constraining either the model's generality or a project's flexibility. [Ref. 3: p. 521] The TRW (SCEP) model uses a standard work breakdown structure to define costs included and excluded in estimates.

b. Fidelity

COCOMO estimates come within 20% of the actual development figures for the projects in the COCOMO database 70% of the time. This means a standard deviation of the residuals of roughly 20% of the actuals. [Ref. 3: p. 521] An analysis of the IBM-FSD model reported a standard deviation of 1.71 [Ref. 48: p. 521]. The Bailey-Basili showed a standard deviation factor of 1.15 for a fairly uniform set

TABLE VI

Factors Used in Various Cost Models

Group	Factor	SDC, 1965	TRW, 1972	Putnam, SLIM	Doty	RCA, PRICE S	IBM	BOEING, 1977	GRC, 1979	COCOMO
Size attributes	Source instructions			x	x		x	x		x
	Object instructions	x	x		x	x				
	Number of routines	x				x				
	Number of data items						x			x
	Number of output formats								x	
	Documentation				x		x			
	Number of personnel						x	x		
Program attributes	Type	x	x	x	x	x	x	x		
	Complexity		x			x	x			x
	Language	x		x				x	x	
	Reuse			x		x		x	x	x
	Required reliability					x				x
Computer attributes	Time constraint		x	x	x	x	x	x	x	x
	Storage constraint			x	x	x	x		x	x
	Hardware configuration	x				x				
	Concurrent hardware development	x			x	x	x			x
Personnel attributes	Personnel capability					x	x			x
	Personnel continuity						x			
	Hardware experience	x		x	x	x	x		x	x
	Applications experience		x	x		x	x	x	x	x
	Language experience			x		x	x		x	x
Project attributes	Tools and techniques			x		x	x	x		x
	Customer interface	x					x			
	Requirements definition	x			x		x			
	Requirements volatility	x			x	x	x		x	x
	Schedule			x		x				x
	Security						x			
	Computer access			x	x		x	x		x
	Travel/rehosting	x			x	x				

[Ref. 3: p. 511]

of 18 projects at NASA/Goddard [Ref. 46: p. 115.]. The fidelity of the COCOMO model with respect to the actual costs of projects in the database is better than other models' estimates of those costs. A large portion of the database was used to calibrate the model's parameters. No future projects have yet been completed to evaluate the goodness of the model's estimates.

The Putnam 1978 model gives extreme overestimates on small projects and estimates large projects reasonably well. Putnam's more recently developed SLIM

model appears to have overcome this problem. [Ref. 49: p. 196] The TRW (SCEP) model is still in an experimental state and needs more comparisons of SCEP estimates with actual project results [Ref. 49: p. 198].

c. Objectivity

The SLIM and the Price-S models have made some progress in expanding a single complexity factor into a number of constituent elements [Ref. 3: p. 522]. The original Price-S model was extremely sensitive to the subjective complexity factor [Ref. 49: p. 198]. The COCOMO model has tried to make the complexity factor more objective in a number of ways. Complexity has been made a module level instead of a subsystem or system level rating. Sources of productivity have been separated from the complexity cost drivers as much as possible and made into separate cost drivers. A rating scale for each complexity rating has been developed. The TRW (SCEP) model includes a complexity factor. The complexity rating is a characteristic of each unit in the software, and a complexity scale is available to provide a unique complexity rating for each type of unit.

d. Constructiveness

The COCOMO model provides a detailed listing of the factors affecting the cost of a project. It estimates the impact of an individual factor. The model provides increased understanding of the software lifecycle for the project. [Ref. 3: p. 522] The TRW (SCEP) model provides a scale to indicate the degree of impact of factors on project activities.

e. Detail

Models requiring more detail usually produce more accurate estimates.

1. The gathering of greater detail tends to increase people's understanding of the job to be done; and
2. if the added detail results in the overall estimate being the sum of some smaller individual estimates, the law of large numbers tends to work to decrease the variance of the estimate. [Ref. 49: p. 200]

COCOMO is a hierarchy of models with the Basic COCOMO being used for early estimates and the Intermediate and Detailed COCOMO's being used for more detailed and accurate estimates. The TRW-Wolverton model is an effective micro model and provides detail in phase and activity breakdowns. The 1979 GRC model also provides detail in phase and activity breakdowns. [Ref. 3: p. 522]

f. Stability

The Doty model has discontinuities at the neighborhood of 10,000 source instructions. Small differences in sizing can lead to large differences in cost in this area. [Ref. 50] Most cost estimating models, COCOMO included, avoid this problem by providing a number of rating levels for cost driver attributes and allowing interpolation between them.

g. Scope

The IBM-FSD model, the Meta model, the Price-SL and the COCOMO models have all been developed to meet a wide variety of projects and applications. Algorithmic cost models in general have a difficult time in general in estimating cost for projects under 2000 DSI. [Ref. 3: p. 523]

h. Ease of Use

SLIM and Price-S are well engineered for ease of use and understanding. COCOMO hierarchy of models makes them easy to use and to understand. [Ref. 3: p. 523]

The TRW (SCEP) model overestimates costs on projects less than five person years in total effort, but it functions well for projects over the range of 60-2000 manmonths.

i. Prospectiveness

Most current cost models including COCOMO use parameters that can be estimated rather well at the beginning of a project. The only exception for COCOMO is the difficulty with sizing the project.

j. Parsimony

The Walston-Felix model uses different entries for modern programming practices where one would be alright for practical estimation of projects. [Ref. 48] The COCOMO model makes efforts to only use factors that have a considerable affect on software productivity. The model can be tailored to a particular environment to eliminate redundancy in factors. [Ref. 3: p. 524]

B. ESTIMATING COST AND EFFORT: CRITICAL FACTORS

1. Discussion

We conclude that what is needed in the field of estimating cost and effort in software development projects is a reliable, dynamic, transportable model that is easy to use. It appears intuitively obvious to us that cost, effort and time can be saved by adopting an already existing cost and effort estimating model to a new environment rather than

generating an entirely new model from the ground up. The model should be able to estimate cost and effort throughout the lifecycle of a software project. Most models now only estimate through the completion of testing and the beginning of operation giving little or no attention to the maintenance phase. The maintenance phase of a software lifecycle currently consumes the major portion of resources expended upon a software development effort [Ref. 34: p. viii].

Any measure of effort should be linked to the successful completion of the functions of a project. The preliminary work on a particular design decision may be well understood by software developers. The basic steps may account for a major physical portion of the effort. The concluding work done to implement a design decision and the integrating of numerous design decisions/modules to make the system operational often commands the greatest effort. The model should measure effort in the number of lines of source code (LOSC) produced but should also relate this figure to the area of applicability of the lines. To reiterate, LOSC produced at the beginning of the development of a design decision may be far easier to produce than those at the end of the effort.

Statistical investigation should be used to establish relationships which make it possible to predict cost and effort in terms of other variables. Regression techniques are used to perform this task. Since the number of variables affecting the cost and effort estimated for a given project will be many, multiple regression analysis will be necessary. In using observed data to formulate a mathematical equation to predict desired values from given values (a procedure known as curve fitting), three problems arise:

1. the kind of equation to be used must be decided

2. the best of this type must be found
3. the goodness of fit of the equation must be determined.

[Ref. 51: pp. 431-433]

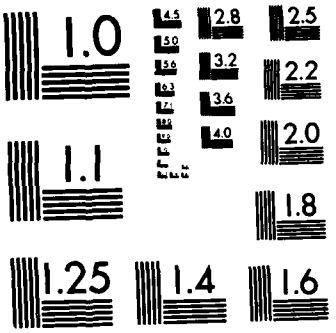
The equation usually chosen results from the inspection of the data in most instances, but the most objective methods for deciding on what curve to fit to numerous points should be used. Differences in project estimations will be explained in accordance with environmental variations. The key to estimating cost and effort in a software development project is to isolate those elements that cause project estimates to differ from expected values. Once these elements are identified, they can be accounted for in the estimation process and extremely accurate estimates can be achieved.

C. SUMMARY

We have endeavored to present a number of environmental factors influencing software development projects, and the methods now in use to predict cost and effort for those projects. From our study of the literature in the area of software development and from an analysis of various models, we have tried to assimilate those problems that should be addressed in the development of a dynamic, transportable, prediction model. We also endeavored to alert the novice to and refresh the experienced reader with the problems he may expect to encounter with software cost and effort estimation models. As it is probably painfully apparent to the reader at this point, models are often complex and difficult to understand. We recommend to the average manager that he familiarize himself with the information presented in the research and then go and hire someone who is technically competent for specific guidance. If it is of any

AD-A126 358 SOFTWARE DEVELOPMENT PROJECTS: ESTIMATION OF COST AND EFFORT (A MANAGER'S DIGEST)(U) NAVAL POSTGRADUATE SCHOOL MONTEREY CA C J PIERCE ET AL. DEC 82 2/2

UNCLASSIFIED F/G 9/2 NL



compensation to the reader, the authors of this research have had as difficult a time as he or she may have had in understanding the models presented.

The key to the success of any such model is the ability of the estimators to identify variables in the environment affecting the estimations and account for these variables in the mathematical equation predicting cost and effort. The weaknesses with current estimating procedures are sixfold:

1. estimating size
2. determining environmental influences
3. determining complexity
4. understanding the models themselves
5. lack of attention to the experience of the developers
6. lack of attention to the management effort and the project manager.

A hopeful avenue of research that may provide more reliable estimates is Silver's method of using structural decomposition of requirements and design parameters. What is missing or under emphasized in most proposals for estimating cost and effort in software development projects in private industry is consideration of the management effort and the project manager. Unless a sound team is organized under a strong leader, all estimations of project cost and effort will prove to be under estimates.

D. THE FUTURE OF SOFTWARE DEVELOPMENT PROJECTS

Estimating cost and effort in software development projects has already been influenced by the introduction of various tools and the concept of software development environments. Programmer productivity is expected to increase as tools are refined and better integrated with one another. What is especially exciting in the long term future of programming, that is, programming into the early decades of the 21st century, is the concept of automatic programming.

> The term 'automatic programming' has been used for many years to refer to the process by which an executable program may be produced from nonprocedural specifications of the task to be performed. Over the longer term, it will be possible for programmers to create running programs by providing a specification of program functions and outputs, without having to proceed with a detailed program design or with the production of code. [Ref. 52: p. 204]

The present differences between application programmers and system programmers are likely to increase. System programmers deal with the details of the low level computer whereas application programmers deal with the development of programs to meet user specifications. With the anticipated advent of automatic programming, the user-operator will carry out what we now consider programming as he interacts using natural language with the computer. The application programmer will increasingly be involved with understanding the needs of particular application areas for software, i.e., medical and information system applications, their information requirements, organizational structures and their personnel makeup. He will assist the user-operators of the companies in understanding their needs and converting these needs into specific requests to be automatically programmed by the computer into an affective application software program.

> ...it can be seen that the nature of programming and programmers is certain to change, and that an increasing share of what we now term programming will be carried out by user-operators, who will have tools at their disposal that permit them to interact naturally with a computer system and specify their requests. It is only when such tools are provided that the exponential growth in the number of programmers and the cost of software can be slowed and that attention may be devoted to making the greatest possible beneficial use of the computer. [Ref. 52: p. 205]

LIST OF REFERENCES

1. Forman, J.J., "How Much Does Configurement Management Cost?," Software Engineering Standards Application Workshop, Proceedings, 18-20 Aug., 1981, pp. 41-44.

2. Mohanty, S.N., "Software Cost Estimation: Present and Future," Software---Practice and Experience, Vol. II, 1981, pp. 103-121.

3. Boehm, B.W., Software Engineering Economics, Englewood Cliffs, New Jersey: Prentice Hall, Inc., 1981.

4. Peters, L. J. & Tripp, L. L., "A Model of Software Engineering," 3rd International Conference on Software Engineering, Proceedings, 10-12, May, 1978, pp. 63-70.

5. Remus, H., "Planning and Measuring Program Implementation," Software Engineering Environment, Proceedings, 16-20 June, 1980, pp. 223-235.

6. Kerola, P. & Freeman, P., "A Comparison of Lifecycle Models," 5th International Conference on Software Engineering, Proceedings, 1981, p. 90-91.

7. Brooks, F.P., The Mythical Man-Month, Phillipines: Addison-Wesley Publishing Company, Inc., 1982.

8. Esterling, B., "Software Manpower Costs: A Model," Datamation, March, 1981, pp. 164-170.

9. Schneider, G.M., Sedlmeyer, R.L. & Kearney, J., "On the Complexity of Measuring Software Complexity," AFIPS Conference Proceedings, Proceedings, 4-7 May, 1981, pp. 317-322.

10. Christensen, K., Fitsos, G.P. & Smith, C.P., "A Perspective on Software Science," IBM System Journals, vol. 20, no. 4, 1981, pp. 372-387.

11. Spier, M.J. & Gutz, S., "The Ergonomics of Software Engineering," Software Engineering Environments, Proceedings, 16-20 June, 1980, pp. 223-234.

12. Turban, E. & Meredith, J.R., Fundamentals of Management Science, Plano, Texas: Business Publications, Inc., 1981, pp. 271-311.

13. Roberts, E., The Dynamics of Research and Development, New York: Harper & Row, 1964, pp. 34-52.

14. Federal Information Processing Standards Publication 76, "Guideline for Planning and Using a Data Dictionary System," 20 August 1980.

15. Prentice, D., "An Analysis of Software Development Environments," ACM SIGSOFT SOFTWARE ENGINEERING NOTES, October, 1981, pp. 19-27.

16. Boehm, B.W., "Software and Its Impact: A Quantitative Assessment," cited by Prentice, D., in "An Analysis of Software Development Environments," ACM SIGSOFT SOFTWARE ENGINEERING NOTES, October, 1981, pp. 19-27.

17. Stoner, A.F., Management, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1982.

18. Duncan, C.S., "The Challenges Facing Program Managers," Program Managers Newsletter, October, 1977, pp. 8-9.

19. Caron, P.F. & Roderick, B., "The Challenge of Program Management: Building and Motivating a Team," Government Contracts Service, November 15, 1979, pp. 6-9.

20. Smith, G.A., "Some Thoughts on the Art of Motivation," Program Managers Newsletter, March, 1976, p. 19.

21. Acker, D.D., "Managing Creativity and Innovation," Program Managers Newsletter, Summer, 1976, p. 15.

22. Blair, R.S., Colonel, USAF, "Managers Are You Really Listening To Your Employees?" Program Manager, May-June, 1981, pp. 4-6.

23. Hussain, D. & Hussain, K.M., Information Processing Systems for Management, Homewood, Illinois: Richard D. Irwin, Inc., 1981.

24. Bostrom, R.P. & Heinen, J.S., "MIS Problems and Failures: A Socio-Technical Perspective PART I: THE CAUSES," MIS Quarterly, Sept., 1977, pp. 17-22.

25. Zelkowitz, M.V., "Perspectives on Software Engineering," ACM Computing Surveys, June, 1978, pp. 197-215.

26. Schneidewind, N.F., Software Maintenance: Improvement Through Better Development Standards and Documentation, Monterey, Calif.: Naval Postgraduate School, 1982.

27. Young, R.A., "Life Cycle Concepts and Document Types," in Software Maintenance: Improvement Through Better Development Standards and Documentation, Monterey, Calif.: Naval Postgraduate School, 1982.

28. Liskov, B.H., "A Design Methodology for Reliable Software Systems," Fall Joint Computer Conference, Proceedings, 1972, pp. 65-73.

29. Stevens, W.P., Myers, G.J. & Constantine, L.L., "Structured Design," IBM Systems Journal, Vol. X, No. 3, 1974, pp. 216-224.

30. Parnas, D.L., "On the Criteria to be Used in Decomposing Systems into Modules," Communications of the ACM, December, 1972, pp. 178-183.

31. Parnas, D.L., "Designing Software for Ease of Extension and Contraction," IEEE Transactions on Software Engineering, March, 1979, pp. 226-235.

32. Parnas, D.L., "Information Distribution Aspects of Design Methodology," in "A Design Methodology for Reliable Software Systems," Fall Joint Computer Conference, Proceedings, 1972, pp. 65-73.

33. Mader, C., Information Systems, Chicago: Chris Mader, 1979, pp. 308-312.

34. McClure, C.L., Managing Software Development and Maintenance, New York: Van Nostrand Reinhold Company, 1981.

35. Wolverton, R.W., "The Cost of Developing Large-Scale Software," IEEE Transactions on Computers, Vol. C-23, No. 5, June, 1974, pp. 615-636.

36. Lecht, C.P., The Management of Computer Programming Projects, New York: American Management Association, Inc., 1967.

37. Frick, R.K., "Viewing Cost as a Management Tool" National Aerospace and Electronics Conference, Proceedings, 19-21 May, 1981, pp. 888-890.

38. Barakat, D.H., "Productivity and the Development Environment," Proceedings of the IEEE COMPCON, Fall 1981, p. 245.

39. Kiser, B.C. Stewart, "Software Management Productivity Understanding the Software Development Process" Proceedings of the IEEE COMPCON, Fall, 1981, p. 244.

40. Stone, J., "Productivity Measures Prove Counterproductive," Computerworld, 1 September, 1980, p. 29.

41. Wiener-Ehrlich, W.K., Hamrick, J. & Rupolo, V., "Applicability of the Rayleigh Model to Three Different Types of Software Projects," Proceedings of the IEEE COMPCON, Fall, 1981, pp. 128-148.

42. Bruce, P. & Pederson, S.M., The Software Development Project, Planning and Management, New York: John Wiley and Sons, 1982, pp. 16-24.

43. Thibodeau, R. & Dodson, F. N., "The Implications of Life Cycle Phase Interrelationships for Software Cost Estimating," Proceedings of the Second Software Lifecycle Management Workshop, Atlanta, Georgia, 21-22 August, 1978, pp. 70-76.

44. Dircks, H.F., "'SOFCOST' Grumman's Software Cost Estimating Model," National Aerospace and Electronics Conference, Proceedings, 19-21 May, 1981, pp. 678-682.

45. Silver, A.N., "Software Life Cycle Cost (LCC) Estimating Using Structural Decomposition of Requirements and Design Parameters," National Aerospace and Electronics Conference, Proceedings, 19-21 May, 1981, pp. 665-672.

46. Bailey, J. W., & Basili, V. R., "A Meta-Model for Software Development Resource Expenditures," Fifth International Conference on Software Engineering, Proceedings, 9-12 March, 1982, pp. 107 - 116.

47. Mauro, C., "RCA Price System," National Aerospace and Electronics Conference, Proceedings, 19-21 May, 1981, pp. 663-664.

48. Walston, C.E. & Felix, C.P., "A Method of Programming Measurement and Estimation," cited by Boehm, B.W. in Software Engineering Economics, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1981.

49. Boehm, B.W. & Wolverton, R.W., "Software Cost Modeling: Some Lessons Learned," The Journal of Systems and Software, Dec., 1980, pp. 195-201.

50. Boehm, B.W. & Wolverton, R.W., "Software Cost Modeling: Some Lessons Learned," cited by Boehm, B.W. in Software Engineering Economics, Englewood Cliffs, New Jersey, Prentice-Hall, Inc., 1981.

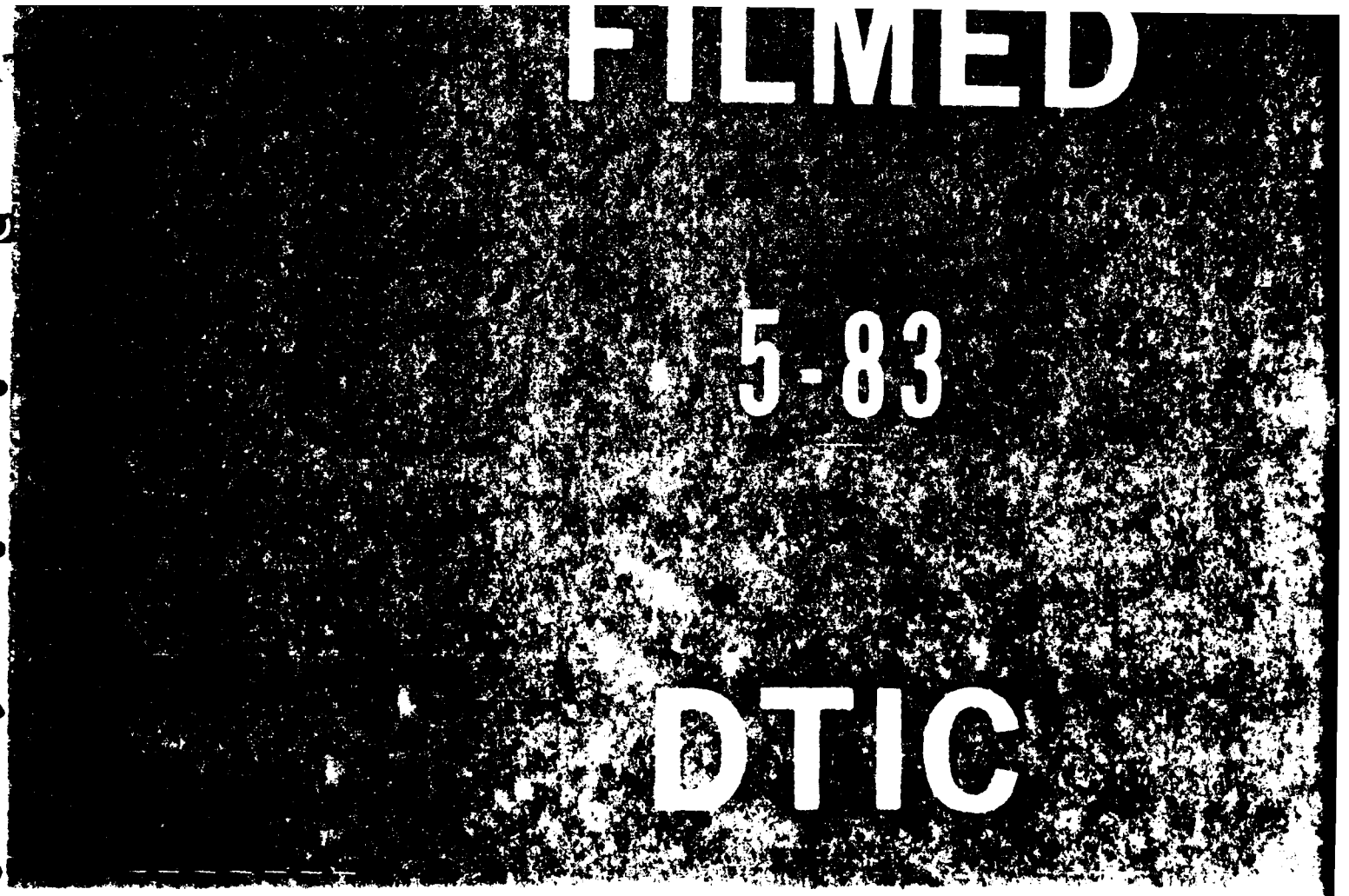
51. Freund, J.E. & Williams, F.J., Elementary Business Statistics: The Modern Approach, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1982, pp. 431-454.

52. Wasserman, A.I. & Gutz, S., "The Future of Programming," Communications of the ACM, March, 1982, pp. 196-205.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Department Chairman, Code 59 Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	1
4.	Curricular Office, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93940	1
5.	Captain Bradford D. Mercer, USAF Code 52ZI Department of Computer Science Naval Postgraduate School Monterey, California 93940	5
6.	Associate Professor Weissinger-Baylon Code 54WR Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	3
7.	Lieutenant Chuck Pierce, USN 4383 Hydrangea Court San Diego, California 92154	3
8.	Lieutenant Rebecca L. Wagner, USN 98-926A Iho Place Aiea, Hawaii 96701	1
9.	Vice Admiral G.R. Nagler, USN CNO (OP-094) Department of the Navy Washington, D.C. 20350	1
10.	Rear Admiral P.E. Sutherland, USN Naval Data Automation Command Washington Navy Yard Washington, D.C. 20374	1
11.	Captain A.H. Fredrickson, USN CNO (OP-942) Department of the Navy Washington, D.C. 20350	1
12.	Dr. Joel S. Lawson Code 06T Naval Electronics Systems Command Department of the Navy Washington, D.C. 20360	1

13.	Lieutenant Commander Ronald Modes, USN Code 52MF Department of Computer Science Naval Postgraduate School Monterey, California 93940	2
13.	Mr. & Mrs. Lew Zuber 992 Church Street Bohemia, Long Island, New York 11716	1
14.	Mr. & Mrs. Charles J. Pierce 138 Connecticut Avenue Massapequa, Long Island, New York 11758	1
15.	Mrs. Renetta M. Lynch 14512 Chesterfield Road Rockville, Maryland 20853	1


FILMED
5-83
DTIC